OUTI RÄIHÄ

Genetic Algorithms in Software Architecture Synthesis

ACADEMIC DISSERTATION
To be presented, with the permission of
the board of the School of Information Sciences
of the University of Tampere,
for public discussion in the Auditorium Pinni B 1097,
Kanslerinrinne 1, Tampere,
on November 18th, 2011, at 12 o'clock.

UNIVERSITY OF TAMPERE

UNIVERSITY OF TAMPERE

ACADEMIC DISSERTATION
University of Tampere
School of Information Sciences
Finland

Distribution
Bookshop TAJU
P.O. Box 617
33014 University of Tampere
Finland

Tel. +358 40 190 9800
Fax +358 3 3551 7685
taju@uta.fi
www.uta.fi/taju
http://granum.uta.fi

Cover design by
Mikko Reinikka

Acta Universitatis Tamperensis 1654
ISBN 978-951-44-8562-6 (print)
ISSN-L 1455-1616
ISSN 1455-1616

Acta Electronica Universitatis Tamperensis 1115
ISBN 978-951-44-8563-3 (pdf)
ISSN 1456-954X
http://acta.uta.fi

Tampereen Yliopistopaino Oy – Juvenes Print
Tampere 2011

Abstract

This thesis presents an approach for synthesizing software architectures with genetic algorithms. Previously in the literature, genetic algorithms have been mostly used to improve existing architectures. The method presented here, however, focuses on upstream design. The chosen genetic construction of software architectures is based on a model which contains information on functional requirements only. Architecture styles and design patterns are used to transform the initial high-level model to a more detailed design. Quality attributes, here modifiability, efficiency and complexity, are encoded in the algorithm's fitness function for evaluating the produced solutions. The final solution is given as a UML class diagram. While the main contribution is introducing the method for architecture synthesis, basic tool support for the implementation is also presented.

Two case studies are used for evaluation. One case study uses the sketch for an electronic home control system, which is a typical embedded system. The other case study is based on a robot war game simulator, which is a typical framework system. Evaluation is mostly based on fitness graphs and (subjective) evaluation of produced class diagrams.

In addition to the basic approach, variations and extensions regarding crossover and fitness function have been made. While the standard algorithm uses a random crossover, asexual reproduction and complementary crossover are also studied. Asexual crossover corresponds to real-life design situations, where two architectures are rarely combined. Complementary crossover, in turn, attempts to purposefully combine good parts of two architectures.

The fitness function is extended with the option to include modifiability scenarios, which enables more targeted design decisions as critical parts of the architecture can be evaluated individually. In order to achieve a wider range of solutions that answer to competing quality demands, a multi-objective approach using Pareto optimality is given as an alternative for the single weighted fitness function. The multi-objective approach evaluates modifiability and efficiency, and gives as output the class diagrams of the whole Pareto front of the last generation. Thus, extremes for both quality attributes as well as solutions in the middle ground can be compared.

An experimental study is also conducted where independent experts evaluate produced solutions for the electronic home control. Results show

that genetic software architecture synthesis is indeed feasible, and the quality of solutions at this stage is roughly at the level of third year software engineering students.

Tiivistelmä

Tämä väitöskirja esittelee menetelmän, joka käyttää geneettisiä algoritmeja ohjelmistoarkkitehtuurisynteesissä. Geneettisiä algoritmeja on aiemmin käytetty lähinnä parantamaan olemassa olevia arkkitehtuureja; nyt esitetyssä tutkimuksessa pyritään puolestaan selvästi luomaan uusia arkkitehtuureja. Valittu geneettinen mallinnus perustuu pelkästään toiminnallisiin vaatimuksiin, joiden avulla muodostetaan arkkitehtuurin perusta, n.s. "nolla-arkkitehtuuri". Tätä korkean tason ilmentymää arkkitehtuurista muokataan ottamalla käyttöön arkkitehtuurityylejä ja suunnittelumalleja, jolloin lopputuloksena on huomattavasti yksityiskohtaisempi suunnitelma arkkitehtuurista. Tuotettuja arkkitehtuureja on arvioitu kolmen laatuvaatimuksen suhteen: muunneltavuus, tehokkuus ja ymmärrettävyys. Laatuattribuutteja on mitattu metriikoilla, jotka on koottu genettisen algoritmin hyvyysfunktioon. Lopputulos tuotetaan UML-luokkakaaviona. Vaikka pääpaino on syntetisointiprosessin esittelyssä, esitellään väitöskirjassa myös työkalu, joka tarjoaa peruskäyttöliittymän syntetisoitujen arkkitehtuurien tuottamiseen.

Arvioinnissa on käytetty tapaustarkastelua, jossa on kaksi erilaista järjestelmää. Toinen tapauksista on luonnos e-kotijärjestelmästä, joka on tyypillinen sulautettu järjestelmä. Toinen tapaus perustuu robottisota-pelisimulaattoriin, joka on tyypillinen kehysjärjestelmä. Arvioinnissa on käytetty hyvyysfunktiograafeja sekä (subjektiivista) evaluointia tuotetuista luokkakaavioista.

Geneettisen algoritmin perustoteutuksen risteytykseen ja hyvyysfunktioon on kehitetty erilaisia parannuksia. Perusalgoritmin käyttäessä satunnaista risteytystä kokeita on tehty myös aseksuaalisella lisääntymisellä sekä täydentävällä risteytyksellä. Aseksuaalinen lisääntyminen kuvaa parhaiten arkkitehtuurisuunnittelun todellisuutta, sillä on erittäin harvinaista, että kaksi kilpailevaa arkkitehtuurisuunnitelmaa yhdistettäisiin satunnaisesti. Täydentävä risteytys puolestaan on suunniteltu tilanteisiin, jossa yritetään yhdistää kahden eri arkkitehtuurin parhaat puolet.

Hyvyysfunktiossa voidaan myös huomioida skenaariot, jotka mahdollistavat kohdennettujen suunnitteluratkaisujen käytön tunnistamalla arkkitehtuurin kriittisiä osia ja auttamalla niiden yksityiskohtaisemmassa arvioinnissa. Skenaarioiden käytön lisäksi esitellään vaihtoehtoinen, monioptimoiva versio hyvyysfunktiolle. Tämä Pareto-optimaalisuutta käyttävä hyvyysfunktio pystyy tuottamaan laajan

skaalan erilaisia ratkaisuja, jotka täyttävät eri laatuvaatimuksia. Monioptimoiva menetelmä arvioi erikseen muunneltavuutta ja tehokkuutta, ja antaa tuloksena koko Pareto-rintaman ratkaisut luokkakaavioina. Täten voidaan helposti vertailla ääriratkaisuja kummankin laatuvaatimuksen suhteen perusmenettelyllä saavutettujen keskivertoratkaisujen lisäksi.

Väitöskirjassa esitellään myös kokeellinen tutkimus, jossa riippumattomat asiantuntijat ovat arvioineet automaattisesti tuotettuja (syntetisoituja) ratkaisuja e-kotijärjestelmälle. Tuloksista voidaan päätellä, että geneettisellä algoritmilla syntetisoidut ohjelmistoarkkitehtuurit ovat yhtä hyviä kuin kolmatta vuotta opiskelevien ohjelmistotuotannon opiskelijoiden tuottamat arkkitehtuurit.

Acknowledgements

Writing this thesis has been a consuming process. Performing the research which provided the content for the thesis has been an even more consuming process. The research has consumed much time, energy, emotional and spiritual assets and sometimes even physical efforts when sitting by the computer has turned my shoulders rock hard. But it has all been worth every effort, as I have learned more than I could ever have hoped during this process.

Being such a strenuous process, I could not have done the research and written the thesis without all the support that I have received. First of all, I would like to deeply thank my supervisor, Professor Erkki Mäkinen. He has always been ready to answer my questions and provide feedback. Most importantly, I could always count on his support, and during times when work was about to take too much of a hold of my life, he reminded me to take it easy and that great science was not made by an overworked mind. I was lucky to have two wonderful supervisors and thus, I also owe my greatest appreciation to Professor Kai Koskimies, whose guidance has been invaluable. He has showed such faith in me and offered words of encouragement when most needed. He also generously provided me with a researching position in a project where I could completely focus on my doctoral research, and gain a broad insight to the research community by attending a multitude of conferences. I would also like to give special thanks to both of my supervisors for all those inspirational meetings at Plevna.

I am grateful to Professor Tarja Systä, who showed such great support during the very beginning of my doctoral research, when I was still trying to get a foothold in the research community. I am indebted to my co-authors, M.Sc. Hadaytullah and M.Sc. Sriharsha Vathsavayi, especially for their input to the Darwin tool. They gave my Frankenstein a face that could be shown to the world.

I thank the pre-examiners of my thesis, Eila Ovaska, Ph.D., and Professor Mark Harman, as their comments greatly helped in improving this thesis. I also want to thank my opponent, Professor Jukka Paakki for reviewing this thesis. I give my appreciation to SoSE graduate school for their support and for providing means to get valuable feedback early on in my research. The research in this thesis was mostly done in the Darwin project, funded by the Academy of Finland. In addition to SoSE, the work was also supported by the Nokia Foundation and Tampereen Yliopiston Tukisäätiö.

I want to give my thanks to the administrative staff of the former Department of Computer Sciences at the University of Tampere. I would also like to thank especially the secretarial staff at the Department of Software Systems at Tampere University of Technology, who have provided me with a friendly working environment and helped with all the practicalities that are often so difficult for a researcher with a wandering mind.

While I have enjoyed such great support from my research and work community, this work would not have been possible without my support network at home. I give my deepest appreciation to my whole extended family (spouses included) for their support during my studies. My biggest, heartfelt thanks go to my mother Liisa and my father Kari-Jouko, who have helped and supported me in so many different ways. I also want to acknowledge my sisters Satu and Sini for helping me to perform a total escape from work every now and then. Sometimes one just needs to drop the role of grad student and researcher, and adopting the role of big sister leaves little room for anything else.

I have been lucky to get warm and understanding people as my soon-to-be in-laws. I give my warmest thanks to Sirpa and Kari Sievi-Korte for making their home my home as well, and providing me with everything I could have possibly needed when approaching deadlines forced me to drag work along during the holidays.

Besides family, I have two amazing friends who have always been there for me. I am indebted to Outi, who has shared with me this rewarding but rocky path through graduate studies. She has provided realtime feedback to the little problems that I did not want to go to the professors with, and bounced ideas with me. I am eternally grateful especially for all the help during those times when Excel just wouldn't be my buddy. I also want to thank Petra, who has always believed that one day I will become a Doctor, and provided me with opportunities to let my hair down and totally forget about work.

Finally, my heartfelt thanks go to my beloved fiancé Juha. You have been my partner to celebrate with at times of success, provided a shoulder to cry on at times of failure, and listened to all my hopes, doubts and plans during all the times in between. You have been the one to pick me up from the airport in the middle of the night and rub my aching shoulders. You have made sure that I could concentrate on work when needed, and relax when needed. You have brought light and balance into my life, and I could not have been able to do this without you. Thank you.

Contents

1 INTRODUCTION ... 18

1.1 RESEARCH CONTEXT ... 18
1.2 RESEARCH QUESTIONS ... 20
1.3 THE GA APPROACH ... 21
1.4 RESEARCH METHOD ... 22
1.5 CONTRIBUTIONS AND OVERVIEW ... 24

2 BACKGROUND ... 26

2.1 SOFTWARE ARCHITECTURES ... 26
2.1.1 Software architecture design process ... 27
2.1.2 Software architecture styles and design patterns ... 31
2.1.3 Software architecture quality metrics ... 34
2.1.4 Software architecture evaluation ... 37
2.2 GENETIC ALGORITHMS ... 39
2.2.1 Encoding ... 40
2.2.2 Mutations ... 41
2.2.3 Crossover ... 42
2.2.4 Fitness ... 43
2.2.5 Selection ... 44
2.2.6 Parameters ... 45
2.3 SEARCH-BASED SOFTWARE DESIGN ... 46
2.3.1 Class design ... 47
2.3.2 Low-level architecture transformations ... 47
2.3.3 High-level architectural transformations ... 48
2.3.4 Comparison to presented work ... 49

3 GENETIC SOFTWARE ARCHITECTURE SYNTHESIS ... 51

3.1 METHOD ... 51
3.1.1 Input ... 52
3.1.2 Encoding ... 57
3.1.3 Mutation and crossover ... 58
3.1.4 Fitness and selection ... 65
3.1.5 Case studies ... 69
3.2 TOOL SUPPORT ... 73

4 VARIATIONS AND EXTENSIONS ... 76

4.1 ASEXUAL REPRODUCTION ... 77
4.1.1 Method ... 77
4.1.2 Case studies ... 78
4.2 COMPLEMENTARY CROSSOVER ... 81
4.2.1 Method ... 81
4.2.2 Case studies ... 83
4.3 SCENARIOS ... 87
4.3.1 Method ... 88
4.3.2 Case studies ... 89
4.4 MULTI-OBJECTIVITY ... 96
4.4.1 Method ... 96
4.4.2 Case studies ... 98

5 EVALUATION 106

5.1 SUMMARY OF RESULTS 106
5.2 EVALUATION OF PARETO FRONTS 108
5.3 EMPIRICAL STUDY 111
5.3.1 Setup 111
5.3.2 Results 112
5.3.3 Threats and limitations 114
5.4 DISCUSSION 115
5.4.1 Input and encoding 115
5.4.2 Mutations 116
5.4.3 Fitness function 118
5.4.4 Case systems 119
5.4.5 Limits and potential 121

6 CONCLUSIONS 123

7 INTRODUCTION TO PUBLICATIONS 127

8 REFERENCES 131

List of Included Publications

This thesis consists of a summary and the following original publications, reproduced here by permission.

I. O. Räihä, K. Koskimies and E. Mäkinen, Genetic Synthesis of Software Architecture, In: *Proc. of the 7th International Conference on Simulated Evolution and Learning (SEAL'08)*, Melbourne, Australia. December 2008, Springer LNCS **5361**, 565-574.

II. O. Räihä, K. Koskimies, E. Mäkinen and T. Systä, Pattern-Based Model Refinements in MDA, *Nordic Journal of Computing*, **14** (4), 2008, 338-355.

III. O. Räihä, K. Koskimies and E. Mäkinen, Scenario-Based Genetic Synthesis of Software Architecture, In: *Proc. of the 4th International Conference on Software Engineering Advances (ICSEA'09)*, Porto, Portugal. September 2009, IEEE CS Press, 437-445.

IV. O. Räihä, K. Koskimies and E. Mäkinen, Empirical Study on the Effect of Crossover in Genetic Software Architecture Synthesis, In: *Proc. of the World Congress in Nature and Biologically Inspired Computing (NaBiC'09)*, Coimbatore, India. December 2009, IEEE Press, 619-625.

V. Hadaytullah, S. Vathsavayi, O. Räihä and K. Koskimies, Tool Support for Software Architecture Design with Genetic Algorithms, In: *Proc. of the 5th International Conference on Software Engineering Advances (ICSEA'10)*, Nice, France. August 2010, IEEE CS Press, 359-366.

VI. O. Räihä, A Survey on Search-Based Software Design, *Computer Science Review*, **4** (4), 2010, 203-249.

VII. O. Räihä, K. Koskimies and E. Mäkinen, Complementary Crossover for Genetic Software Architecture Synthesis, In: *Proc. of the International Conference on Intelligent Systems Design and Application (ISDA'10)*, Cairo, Egypt. December 2010, IEEE Press, 260-265.

VIII. O. Räihä, Hadaytullah, K. Koskimies and E. Mäkinen, Synthesizing Architecture from Requirements: A Genetic Approach, In: P. Avgeriou, J. Grundy, J. G. Hall, P. Lago and I. Mistrik (eds), *Relating Software Requirements and Architectures*, Springer, 2011, ISBN 978-3-642-21000-6, to appear.

IX. O. Räihä, K. Koskimies and E. Mäkinen, Generating Software Architecture Spectrum with Multi-Objective Genetic Algorithms, In *Proc. of the Third World Congress on Nature and Biologically Inspired Computing (NaBIC'11)*, Salamanca, Spain, October 2011, IEEE Press, to appear.

List of Abbreviations

Abbreviation	Description
AADL	Architecture Analysis and Description Language
ADD	Attribute-Driven Design Method
ATAM	Architecture Trade-off Analysis Method
B.Sc.	Bachelor of Science
CBO	Coupling Between Objects
CK	Chidamber and Kemerer
CRA	Class Responsibility Assignment
DIT	Depth of Inheritance Tree
GA	Genetic Algorithm
GP	Genetic Programming
ISO	International Organization for Standardization
LCOM	Lack of Cohesion in Methods
M.Sc.	Master of Science
MDA	Model-Driven Architecture
MOGA	Multi-Objective Genetic Algorithm
NOC	Number Of Children
NOM	Number Of Methods
Ph.D.	Doctor of Philosophy
QADA	Quality-Driven Architecture Design and Quality Analysis
QASAR	Quality Attribute-Oriented Software Architecture Design Method
QMOOD	Quality Model for Object-Oriented Design
QoS	Quality of Service
RFC	Response For Class
SA	Simulated Annealing
SBSD	Search-Based Software Design

UI	User Interface
UML	Unified Modeling Language
WMC	Weighted Methods per Class

List of Figures and Tables

Figure 1, Use case example (ehome) 29
Figure 2, Sequence diagram example (ehome, play music) 29
Figure 3, Mutation 42
Figure 4, Crossover 43
Figure 5, Use case for ehome (adjust temperature) 53
Figure 6, Sequence diagram for adjust temperature use case 53
Figure 7, Class diagram example for ehome (temperature control) 54
Figure 8, Null architecture for ehome 55
Figure 9, Null architecture for robo 56
Figure 10, Supergene sg_i 57
Figure 11, Strategy mutation 59
Figure 12, Adapter mutation 60
Figure 13, Template Method mutation 61
Figure 14, Façade mutation 62
Figure 15, Mediator mutation 62
Figure 16, Client-server mutation 63
Figure 17, Message dispatcher connection mutation 64
Figure 18, Example fitness graph for ehome 68
Figure 19, Fitness curve for ehome 70
Figure 20, Fitness curve for robo 70
Figure 21, Example architecture for ehome 71
Figure 22, Example architecture for robo 72
Figure 23, Screenshot of Darwin tool portraying a fitness graph 74
Figure 24, Screenshot of Darwin tool showing an example solution 75
Figure 25, Fitness curve for ehome, asexual reproduction 79
Figure 26, Fitness curve for robo, asexual reproduction 79
Figure 27, Example solution for ehome, asexual reproduction 80
Figure 28, Example solution for robo, asexual reproduction 81
Figure 29, Gene-selective complementary crossover 83

Figure 30, Fitness curves for ehome, complementary crossover 84
Figure 31, Fitness curves for robo, complementary crossover 85
Figure 32, Fitness curves for ehome, complementary crossover, 750 generations 85
Figure 33, Example solution for ehome, complementary crossover 86
Figure 34, Example solution for robo, complementary crossover 87
Figure 35, Fitness curves for ehome, total fitnesses with and without scenarios 91
Figure 36, Fitness curves for robo, total fitnesses with and without scenarios 91
Figure 37, Scenario fitness curve for ehome 92
Figure 38, Scenario fitness curve for robo 93
Figure 39, Example solution for ehome when scenarios included and overweighted 94
Figure 40, Example solution for robo when scenarios overweighted and replacing general modifiability 95
Figure 41, Initial Pareto fronts for ehome 98
Figure 42, Initial Pareto fronts for robo 99
Figure 43, Final Pareto fronts for ehome 100
Figure 44, Final Pareto fronts for robo 100
Figure 45, Example solution for ehome from modifiable end of Pareto front 101
Figure 46, Example solution for ehome from efficient end of Pareto front 102
Figure 47, Example solution for robo from modifiable end of Pareto front 103
Figure 48, Example solution for robo from efficient end of Pareto front 104
Figure 49, Efficiency scenarios 109
Figure 50, Modifiability scenarios 110
Table 1, Points for synthesized solutions and solutions produced by the students 113
Table 2, Numbers of preferences of the experts 114

1 Introduction

1.1 RESEARCH CONTEXT

Designing software architectures is a critical and highly demanding task. Successful software architects have years of expertise and tacit information, which they have gathered from previous projects and by learning from their mentors. As a result, software architecture design is often considered more like art than a form of engineering. Just like artists can study different painting techniques and how to draw different types of objects, software architects can take advantage of known best practices in architecture design. However, just like a piece of art is much more than just strokes of a brush, and truly only gains value from the artist's own effort, no good architecture can be designed simply by collecting bits and pieces of technical knowledge from the literature. No book can give the ultimate answer to architecture design or how to create beautiful software architectures. While existing literature can provide guidelines and answers to small (technical) sub-problems, combining these guidelines and making sure that the architecture meets all the given functional and, usually conflicting, quality requirements, always requires personal contribution in the end.

While there may be as many methods for software architecture design as there are software architects, the main idea is always the same: the architect starts with a set of functional and quality requirements and produces a design, i.e., a composition of functional components, which implement the functional requirements and satisfy the quality requirements to an acceptable degree. In addition, hardware and implementation platform constraints must be considered, different stakeholders must be consulted, the required effort in coding certain functional solutions must be understood and changes in requirements

must be handled. Bosch [2000] describes the design effort as follows: "The architectural design process can be viewed as a function taking a requirement specification that is taken as an input to the [design] method and an architectural design that is generated as output. However, this function is not an automated process and considerable effort and creativity from the involved software architects is required". Frankel [2003] expresses the same idea while discussing the design process related to MDA, where a functional implementation can be generated from a computational model (architecture): "In an ideal world a requirements analyst would simply submit requirements models to generators that would produce the required systems. In practice, you have to refine a requirements model into a computational model […] that a generator can process.".

The combination of growing complexity in the field of software architecture design and the outcome of designs relying solely on some hidden knowledge does not sound very reliable. Thus, would it not be beneficial, if the function described by Bosch could actually be an automated process? What if the ideal world described by Frankel could be accessible? From Bosch's point of view, which is probably adopted by most software engineers, the answer would most likely be no; such automation would not be possible. However, Frankel's view appears more optimistic. I believe that the ideal world where design can be (partially) automated is reachable to some extent, and so the answer could be yes, if a suitable method is found. I see the answer in meta-heuristic search algorithms, and attempt to seek how far the automation of software architecture design can be taken.

Meta-heuristic search algorithms are used when the search space is too large to travel through with a brute-force algorithm, and no deterministic algorithm can be implemented that would produce a solution within reasonable time. The field of software engineering and especially architecture design is more than suitable for such an algorithm, as there are theoretically innumerous ways of composing architectures for any system. One of the most popular search algorithms is the genetic algorithm (GA). Unlike most search algorithms, GAs are able to perform global searches as they always operate with several solutions at a time and are not confined to working within the boundaries of a fixed neighborhood. Thus, GAs would appear most suitable for such a complex problem as software architecture design.

Current research in the area of applying meta-heuristic search algorithms to software (architecture) design can be roughly divided into four categories: software clustering, systems integration, software refactoring and architecture design. While there are differences on the level of abstraction (e.g., clustering deals with components, refactoring deals with

methods and class structure), what most of the existing studies have in common is that they all require a starting point where the design process is already significantly advanced. In these studies the input is usually a complete design where quality requirements have already been considered at some length. As a result, rather than giving actual proposals to an architect dealing with the design problem, they suggest some minor improvements to the structure or act as confirmation devices for the selected design choices. The solution space is usually also quite limited, as assuming that the initial solution truly is a good one, there is not much an algorithm can do to further optimize the solution in terms of software architecture design. In order to find a completely new search path, the algorithm may actually need to "restart" the search by introducing a large amount of changes that would initially decrease the quality of a solution. However, it would be highly unlikely that a human designer would trust such a suspicious looking algorithm. A simpler starting point, nevertheless, is required if the algorithm is expected to find solutions that would actually give some new ideas for the architect. Thus, an approach requiring only basic information on functional requirements of the system would save the initial design time (no actual architectural design would be required), and give the algorithm a chance for a more thorough traverse through the search space. This kind of approach would thus enable the algorithm to produce solutions the designer might never have thought of, as architectural designs are first and foremost based on the experience of the architect. I will present such an approach in this thesis.

1.2 RESEARCH QUESTIONS

This research is a quest for possibilities. I attempt to find out how far the automation of software architecture design can be taken with GAs. In order to do this, I will examine several subproblems to study how the GA should be optimized for software architecture synthesis. After the GA has been optimized, the quality of produced designs can be evaluated against those designed by humans. The subproblems are formulated below as the research questions this thesis attempts to resolve.

1. What information of the system (and its architecture) under design is required as input for the GA in order to perform the synthesis? I will attempt to use GAs strictly for upstream design instead of merely improving an already validated solution like previous studies do. I also intend to give only the minimal amount of information, and leave as much as possible to the algorithm.

2. How can the architecture be numerically evaluated in the fitness function? The GA is not a human with hidden knowledge, and needs a well-defined formula to calculate the quality of a proposed solution. Are current software metrics sufficient for this, and what kinds of methods are needed to achieve accurate evaluation?

3. Is a traditional and simple GA enough, or should some more complex operations be studied, in order to comply with the problem of software architecture design?

4. How far can the automated design be taken? What is the level of quality that can be achieved with automation? This final question is the most interesting of all.

As a summary, the goal of this research is to discover whether automated software architecture design is possible, and if so, how much background information of the system is required and how should the algorithm be constructed to achieve satisfactory results.

1.3 The GA Approach

The approach presented here for genetic software architecture synthesis uses minimal information of the system that is gathered during specification of requirements. In traditional software design, use cases are a standard starting point for defining the functional requirements for a software system. If the use cases are defined to a detailed enough level, simple responsibilities of a system can be extracted. Further on, refined use cases can be used to straightforwardly produce sequence diagrams, from which classes can be elicited, and responsibilities can now be seen as distinct operations or data entities

After defining the functional requirements for the software system in the way described above, the resulting very basic class diagram is given to the GA. The GA operates based on concepts from biology and evolution. Thus, an encoding for the architecture is required. The presented approach uses an encoding where each operation and the information related to it can be basically encoded as a vector. These operation vectors are then combined and this combination of operations represents the entire architecture. The initial solution where the GA starts the design process is simply the collection of all operations and data entities elicited from use cases, encoded into a form understandable to the algorithm.

The actual design or synthesis of architecture is achieved by introducing standard architectural design decisions, here architecture styles and design patterns, to the initial solution (representing functional requirements), and thus building different, more sophisticated solutions. The solutions are evaluated with a fitness function based on classic software metrics. The metrics are enhanced so that knowledge about the effect of interfaces and other design choices can be taken into account in addition to the simple inheritance and class coupling metrics.

This basic approach, crudely described above, should answer research questions 1 and 2. However, the genetic operators should be further studied in order to answer research question 3. Two variations have been made of the crossover operator, as well as two alternatives or extensions to how the solutions are evaluated and selected. These further studies help determining how the GA should be implemented in order to best resemble the real-life design process and thus produce solutions that would be as good (or better) as a human designer's.

The choice of using only GA could be questioned. Additionally, from a scientific viewpoint, it would be interesting to try a combination of GAs and some other meta-heuristic search method. A local search would either begin the search, in which case the GA would have a significantly improved starting point than the simple null architecture. The other option would be to first use the GA, and then use another algorithm to improve the solution provided by the GA.

Regarding these issues, initial experiments in software architecture synthesis [Räihä et al., 2010] have been made where the GA was combined with simulated annealing (SA) [van Laarhoven and Aarts, 1987]. The preliminary results suggest that simulated annealing could be beneficial in the case where GA is used to produce a solution with good quality, which is then further improved with SA. Simulated annealing is significantly faster than the genetic algorithm, and thus a longer evolution can be performed quicker when part of it can be replaced by the SA. However, curiously enough, tests [Räihä et al., 2010] also show that SA by itself is not capable of producing satisfactory solutions, and applying GA to further improve a solution given by SA does not provide good results either. Thus, the initial decision on using GA for the software architecture synthesis is confirmed with respect to SA.

1.4 RESEARCH METHOD

The research is conducted by implementing a genetic algorithm in the described manner and performing case studies on two different sample systems. The case studies are used to answer research questions one, two and three. As for question four, an experimental study is conducted where

the synthesized solutions are compared with those produced by humans (for the same system).

Case studies are a useful research method when the research questions begin with "how" or "why" [Yin, 2003]. The research questions defined above could all be presented beginning with "how": How should the input be presented; How should the algorithm be defined; How should the evaluation be made; and finally, How good is the algorithm? Case studies are particularly well-suited for theory-forming research, where there are no clear hypotheses in the beginning of the research [Järvinen, 1999], as is the case here. I do not have a particular theory, apart from the optimistic view that meta-heuristic search algorithms could aid in software architecture design, and this research is aimed at finding out how far the synthesis can be taken with GAs.

There are eight steps to performing a case study [Järvinen, 1999]. The first step is to define the research question. The research question should be formed so that there is a clear focus, but it should have as little theory or hypotheses as possible, as they might bias or limit the findings. In this thesis the main research question is, as stated, finding out the possibilities of genetic algorithm based software architecture synthesis.

The second step is selecting the cases. Cases are selected based on the goal of the research. In my research, the goal is to find as general answers as possible, i.e., the algorithm should not be designed for just one type of architectures. Thus, I chose two cases which differ from one another as much as possible.

One case is an electronic home control system. This is a typical embedded system with quite few and large components. There are not many dependencies between different components, and only the user interface and main controller component use several other components. The ehome control system is not expected to be highly efficient, but should be modifiable so that new components (devices in the home) can be easily added.

The other case is a robot war game simulator. This is a typical framework type system, and has several small components. There is more communication between different components than in the case of ehome, and the main controller and user interface have a much smaller role. The robot war game simulator should be efficient as lagging is very undesirable in a gaming application, but also modifiable as components should be easily customized.

The two systems used for the case studies are different both structurally and in what they have as quality requirements. Thus, if the GA is able to produce satisfactory results for both systems, it is reasonable to expect that

the synthesis is, in principle, applicable for various kinds of software systems and architectures.

The third step is crafting instruments and protocols, i.e., determining what kind of data is collected and how. I have collected two kinds of data: the numerical fitness values that the algorithm uses for evaluation as well as the synthesized architectures given as class diagram outputs. This combination of quantitative and qualitative data should provide enough information to evaluate the results from both theoretical and practical viewpoints.

The fourth step is "entering the field" [Järvinen, 1999]. When performing case studies, it is common that gathering and analyzing data overlap. This is also true in the case of this thesis: the research setup and approach was quickly altered according to results from previous data analysis, rather than keeping the same setup for all experiments.

The fifth step is the actual analysis of data. Data is analyzed both within-case and between cases. Within-case analysis gives a clearer understanding how varying certain aspects, e.g., the fitness function, affect the outcome, while between cases analysis provides information of the generality of the approach, and how much the actual system under design affects the outcome rather than the algorithm. The same parameters were used for both cases, while in reality the designer would likely try to optimize the algorithm to suit a certain type of system. By giving the same parameters for the algorithm in both cases it is easier to do between case comparison and see how the particularities of the system under design affect the outcome.

The sixth step is shaping the hypotheses. This is done by analyzing the results from the two cases and refining the method for architecture synthesis accordingly. The seventh step is comparing the theory with existing literature; in my study, this has been done before the actual studies. The final step is bringing the research to closure and providing new concepts, a conceptual framework, propositions or mid-range theory. I will conclude the case study by providing propositions of the possibilities of genetic software architecture synthesis.

1.5 CONTRIBUTIONS AND OVERVIEW

The contributions of this thesis are the following.

1. A novel pattern-based approach using genetic algorithms for software architecture synthesis (publications [I], [II] and [VIII]).

2. A method for encoding modifiability related scenarios as a fitness function for GAs, which resembles human evaluation and enables more detailed design choices (publication [III]).
3. An approach to use asexual reproduction with genetic algorithms in software architecture synthesis, which most resembles actual architecture design, as parts of two architectures are rarely combined (publication [IV]).
4. An approach for purposefully combining two architectures which satisfy different quality requirements. This is implemented as complementary crossover, which is more realistic than randomly combining parts of two architectures (publication [VII]).
5. A thorough review of research in the area of search-based software design (publication [VI]).
6. Preliminary tool support which provides a user interface to the synthesizer (publication [V]).
7. Multi-objective evaluation with Pareto optimality is defined for software architecture synthesis with two objectives, modifiability and efficiency (publication [IX]).

This thesis consists of an introductory part and nine original articles published previously. The introductory part presents background to the research questions and summarizes the studies made in the original publications. The introductory part proceeds as follows: the background and previous studies in this area are presented in Chapter 2. The approach for genetic software architecture synthesis is described in Chapter 3. Studies involving variations and extensions to the basic mechanism are presented in Chapter 4. The different studies are evaluated, summarized and discussed in Chapter 5, and concluding remarks are presented in Chapter 6.

2 Background

In order to understand genetic software architecture synthesis, background information on both the software engineering aspect and the algorithmic aspect is required. In this chapter, the concepts related to software architecture design and its evaluation are first discussed, followed by an introduction to the core concepts of GAs. Finally, existing studies on applying search algorithms to software engineering design problems are discussed.

2.1 SOFTWARE ARCHITECTURES

The core of every software system is its architecture. Designing software architecture is a demanding task requiring much expertise and knowledge of different design alternatives, as well as the ability to grasp high-level requirements and transform them into detailed architectural decisions. In short, designing software architecture takes verbally formed functional and quality requirements and turns them into an architectural model, which is used as a base for implementation. The ISO standard [ISO, 2007] defines software architecture as "the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution". Software architectures can be described by using different views or structures, e.g., module structure (units are work assignments), physical structure (mapping software onto hardware) or data flow (programs or modules are linked with information about the data they exchange) [Bass et al., 1998]. In this thesis I use the class structure to model architecture designs.

The focus in this thesis is on quality-driven software architecture design, which most resembles the GA process. The GA does not consider any

other drivers or stakeholders in the design process than the quality attributes encoded into its fitness function. In traditional software architecture design, there are several quality-driven approaches, e.g., the *quality-driven architecture design and quality analysis* (QADA) method [Matinlassi, 2006; Evesti, 2007] and the *attribute-driven design method* (ADD) [Wojcik et al., 2006]. The genetic synthesis approach presented in this thesis proceeds in a very similar manner as the *quality attribute-oriented software architecture design method* (QASAR) [Bosch, 2000]. This design method consists of three phases: functionality-based architecture design, architecture evaluation and architecture transformation. The functionality-based architecture design roughly corresponds to what is given as input for the genetic algorithm, while the transformations and evaluation are performed iteratively by the algorithm in the form of mutations and fitness evaluation. In QASAR, the design is an iterative process, and two levels of iteration are used. The "outer" iteration considers functional requirements; only a subset of functional requirements can be used in the beginning and other, less critical requirements, can be added later on. The "inner" iteration considers the transformations and evaluation. I will proceed to describe the architecture design process the way it is generally implemented (as according to QASAR) as well as basic steps to achieve the functionality-based architecture design (independent of the selected design method).

2.1.1 Software architecture design process

No matter what methodology is used, software development always follows roughly the following steps: 1) gather requirements, 2) analyze requirements, 3) produce high-level design (architecture), 4) produce code, 5) test, and 6) perform maintenance [Kleppe et al., 2003]. Today, one of the most popular design methodologies, and also the one used in this thesis, is the object-oriented methodology, where steps 1 and 2 of the software development cycle can be combined under one term, *analysis* [Booch, 1991; Rumbaugh et al., 1991]. I assume that this analysis phase is always a manual task and must be done by a human software engineering expert. The transmission from step 2 to step 3, however, can be automated, as I will demonstrate further on.

I will now briefly discuss the design process (steps 1-3), and in particularly how the analysis phase can be performed and what is required to proceed from analysis to architecture. Examples are based on an electronic home control sample system, called hereafter ehome, which is also used as a case study in the genetic synthesis. Firstly, it should be noted, that while in a real-life design situation several business and organizational factors influence the design of software architecture, they are not considered here. Secondly, this description is slightly idealized. A real-life design situation

is unlikely to be as straightforward and would most probably produce many other ways of depicting the requirements, such as other UML diagrams in addition to those mentioned here. Thus, it should be emphasized that the following design process description does not intend to depict an actual situation, but rather to define basic concepts and give enough background to understand what is required from the analysis phase in the case of genetic software architecture synthesis.

The design process in any design method, including QASAR, begins by making a *requirement specification* based on the demands of different stakeholders. The requirements specification contains both the *functional requirements* of a system as well as *quality requirements*. The actual design process for QASAR then proceeds with a subset of the functional requirements (the most critical ones).

In QASAR, the functional requirements are used to construct a *functionality-based architecture design*, which does not yet take into consideration the quality requirements. While the original description for QASAR [Bosch, 2000] specifies a certain methodology to create this rudimentary design, a more simplistic, object-oriented approach is used in this thesis. The outcome, however, is basically the same: an initial view of an architecture, which depicts functional requirements, their mapping to components and the relationships between them, but does not take into account quality requirements.

In this thesis, the following approach for creating the functionality-based design is used. Firstly, in order to formalize functional requirements, the domain must be understood; for this purpose a *conceptual model* of the system under design is created. A conceptual model captures the domain knowledge of the system, and contains the key concepts that can then be used as a basis for eliciting the actual requirements. Thus, once the key concepts are recognized, distinct functional requirements are defined. For this purpose, *use cases* are created. In a use case, the different actors operating with the system are specified, and their actions with the system are defined.

In the case of ehome, we first consider what kind of devices there can be found. Four different devices are specified: heater system, drapes, coffee machine and music system. In addition, there should be some kind of user registry where user info can be set. Based on these, use cases for ehome can describe, e.g., making coffee, setting the room temperature, changing the unit for temperature display, logging in, playing music and moving the drapes. Figure 1 gives the use case diagram for ehome. The PlayMusic case is specified with more detail than other use cases, as it will be used as an example in the following. Other use cases may also include “sub use cases”, but they are not specified in the diagram given in Figure 1.

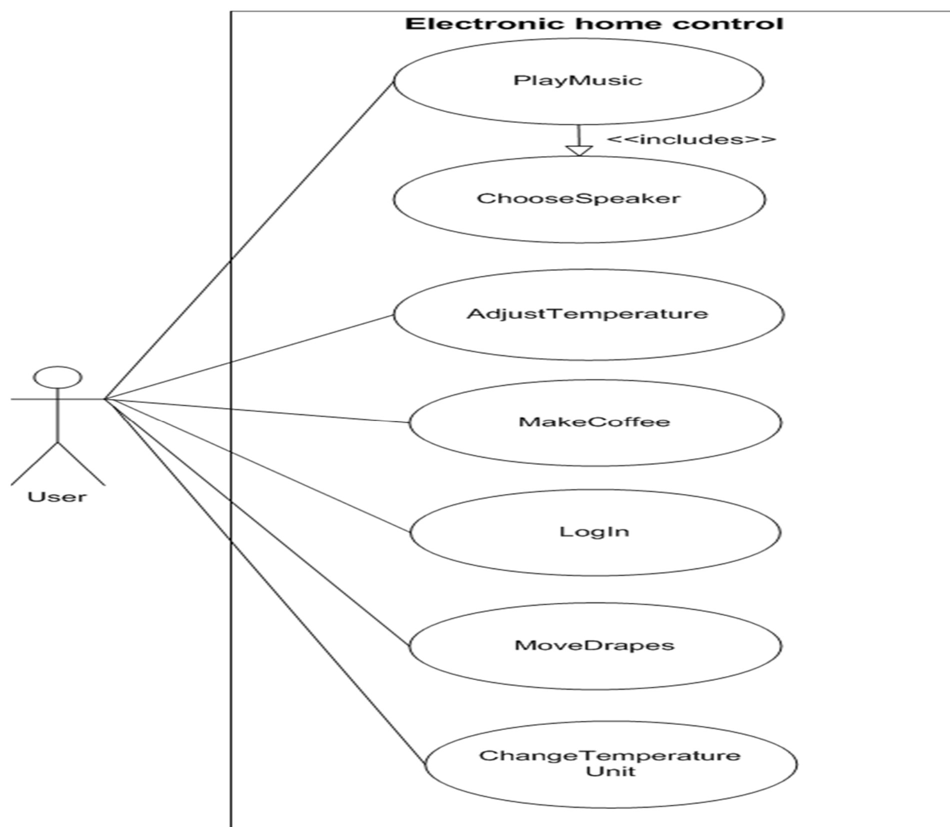


Figure 1. Use case example (ehome)

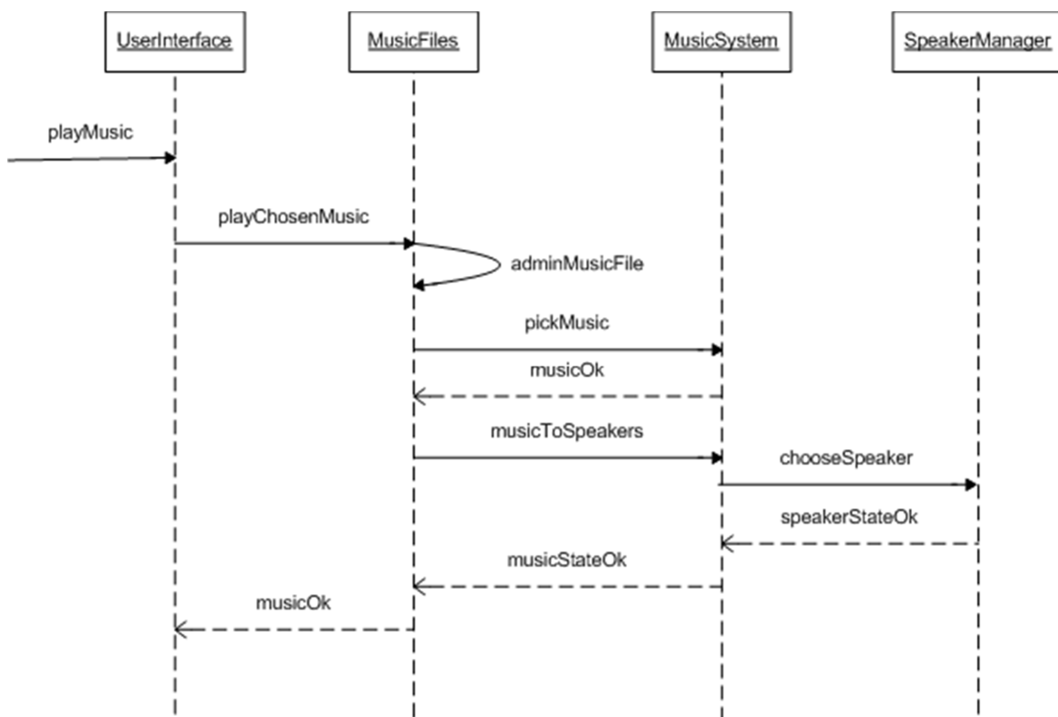


Figure 2. Sequence diagram example (ehome, play music)

Use cases can be refined and transformed into *sequence diagrams*. Sequence diagrams specify the functionality of a system as a sequence of calls (messages) between different objects/classes. Sequence diagrams can easily be extracted from use case diagrams by collecting verbs (messages/calls) and nouns (objects/attributes).

The sequence diagram elicited from the "play music" use case for ehome is given in Figure 2. The process begins with a command from the user to play music. The user interface calls the Music Files component in order to access the selected music file. The Music Files component then calls the Music System in order to actually process the file and use the speakers. The Music System then contacts the Speaker Manager component.

Sequence diagrams give quite a good view already of the underlying software architecture, as they contain all the required objects (components) and the calls (relationships) between them. An architectural view based on sequence diagrams can thus be used as the functionality-based architecture, as required in QASAR. The design process now moves on to consider quality requirements.

Quality requirements usually deal with a set of different *quality attribute*s, such as modifiability, maintainability, performance, usability, portability, and reliability [Bass et al., 1998]. Each quality attribute must be evaluated separately, and they should be balanced. This is a demanding task, as when trying to optimize a design in terms of one quality attribute, one usually has to apply modifications that deteriorate the design in terms of another quality attribute.

In the case of ehome, we can quickly define several critical quality requirements. Firstly, there are requirements related to usability: no one likes such a high-tech home if it is not easily controlled. Secondly, performance is critical: there is no point in automatically moving the drapes if it is slower than the respective manual operation. Thirdly, ehome should be easily extendable to cover new equipment. Finally, the system should be reliable: no one wants to be left outside or be locked in their own home because of a system malfunction.

These quality requirements are still quite abstract, and when analysis progresses, they should be further defined and broken down to more exact requirements. For example, usability related requirements should clearly define what parts of the user interface they concern, and performance related requirements should define limits for lag times. A useful way of further defining quality attributes is to build scenarios for each attribute. Evaluating quality requirements is discussed in more detail further on.

At this point, we have dealt with steps 1 and 2, and are now moving on to step 3 of the software development process: producing an architecture.

The architecture makes sure that the system fulfills the functional requirements while still considering the quality requirements. In QASAR this phase is called *architecture transformation*. The architecture is transformed by applying quality attribute optimizing solutions in order to fulfill the given quality requirements. Bosch [2000] defines architecture optimizing solutions (transformations) as imposing *architecture styles* [Shaw and Garlan, 1996; Bass et al., 1998], imposing *architectural patterns*, applying *design patterns* [Gamma et al., 1995], converting quality requirements to functionality or distributing requirements.

In this thesis, transformations are made by imposing architecture styles and applying design patterns. Applying a certain architecture style usually resolves major issues regarding how communication between components should be handled on a higher level, while design patterns are applied in questions related to a group or just one component or class. Intelligently applying these design choices will eventually produce a software architecture that best answers the given quality requirements. The specific architecture styles and design patterns used in this this study are described in the following subsection.

As stated, QASAR is an iterative process, and at this stage only one iteration has been made. After the architecture has been transformed, it is once again evaluated based on the quality requirements. If they are not satisfied, it is further transformed. This inner iterative process is continued until the quality requirements for the selected functional requirements have been resolved. After this, more functional requirements can be added and the base architecture extended, i.e., the outer iteration is continued. The quality attribute based (inner) iteration thus starts again. The iterative processes of adding functional requirements and then performing evaluations and transformations are repeated until all desired functional requirements are included. The (inner) iterative process of transformations and evaluations closely resembles the way the genetic algorithm processes the software architecture. However, at this stage the GA only works with one iteration of functional requirements, i.e., all functional requirements are given in the beginning of the design process, and the iterative phase only considers adjusting the architecture to meet the quality requirements.

2.1.2 Software architecture styles and design patterns

Software architectures and their design have been studied for decades, and good practices for certain design problems have been identified and documented as architecture styles and design patterns. Design patterns are different for different practices, i.e., object-oriented designs have certain patterns, while service-oriented architectures have other kinds of patterns. Software architectures [Shaw and Garlan, 1996] and object-

oriented design patterns [Gamma et al., 1995] have been collected into catalogues which define proper use for each style and pattern. While the catalogues contain dozens of different styles and patterns, I will here concentrate on the ones used in this thesis.

In this work I use two architecture styles, *message dispatcher* and *client-server*. These architecture styles were selected, as they consider fairly low-level alterations to the architecture. Applying either of these styles requires only structural modifications, and the styles can be visualized at class diagram level, which is essential from the viewpoint of this thesis. Also, applying either architecture style does not require any knowledge of the semantics of different operations.

The message dispatcher style can be seen as an instance of the event-based/implicit invocation style [Shaw and Garlan, 1996]. This architecture style is based on having a messaging component, the dispatcher, in the centre of the architecture. Its only purpose is to receive and send messages between other components. The "basic" components are not communicating directly with each other, but one component merely sends messages to the dispatcher, which then forwards the message to (an)other component(s). There can be direct communication as well, but to get the most out of the message dispatcher style, as many calls between different components as possible should be handled through the message dispatcher. The main benefit of having a dispatcher is independency; components do not need to know anything about each other, they merely need to know what kind of message to send in every different situation. This increases modifiability and flexibility. The drawback is mostly related to efficiency, as having a message dispatcher means that extra time is needed for composing, sending and interpreting messages. Also, complexity increases significantly, as each component communicating with the message dispatcher should implement an interface for receiving messages and a method for sending messages. Furthermore, instead of one call between two components, there are two: one between the sending component and the dispatcher, and another between the dispatcher and the receiving component. The message dispatcher style is commonly used in systems with a large number of components communicating with each other in different ways, and thus a unified communication method is required. Another common use for message dispatcher is in distributed systems, where components are placed in different nodes in a bus, and the nodes then communicate through the dispatcher, while components within the same node communicate directly.

The other architecture style used here is the client-server style [Shaw and Garlan, 1996]. This style is used to encapsulate certain resources. The client merely asks for a certain service from a server, which then provides that service along with the resources. The asking and providing of a

service is usually handled in one session, and thus whatever any other component or server does during that time does not affect the provision of the service. The servers usually also work in their own threads, and thus, are more separated from the clients than traditional classes in object-oriented designs. The servers are usually idle, and merely wait for the clients' requests. The client-server architecture is often used when there is a large database; the database is located in a server, and the components requiring the data become the clients.

In addition to the two architecture styles, I have used five design patterns. While architecture styles can be considered very high-level design choices, patterns can be considered mid-level or low-level design choices. From the design pattern catalogue presented by Gamma et al. [1995] I have used two patterns, *Façade* and *Mediator*, which can be seen as mid-level design choices, and three patterns, *Adapter*, *Strategy* and *Template Method*, which are low-level design choices. These particular patterns were chosen to get samples of both mid-level and low-level design choices. The chosen patterns can be implemented with simple structural modifications, and need not know of any particular semantics of the operations. Thus, as this thesis considers architectures at class diagram level, the chosen patterns are particularly well-suited.

It should be noted that the patterns are only considered from the structural point of view, i.e., how the patterns alter the class diagram and how they improve the architecture. Creating instances of classes (objects) is considered to be a part of the actual implementation and not the high-level architecture, on which this thesis focuses. Thus, object-related concerns in the patterns are omitted or interpreted in terms of classes and/or interfaces.

The Façade pattern is suited for situations where several classes use a group of classes, which also have connections between themselves, and thus the group can be thought of as a subsystem. Thus, in order to decrease coupling, the Façade provides a common interface for other classes to use the subsystem. Technically, the Façade consists of a class and interface: the Façade class requires an interface from all the classes in the subsystem in order to access the actual methods, and the Façade interface is provided for other classes to use the subsystem through the Façade. The Façade increases modifiability by decreasing coupling between classes and hiding information. When the Façade is in place, the calling classes do not even need to know which class will ultimately implement the method they need from the Façade interface.

The Mediator is beneficial in cases where there is a large number of connections between a group of objects (classes). The Mediator is thus used to control and coordinate the interactions of this group and it keeps

the objects (classes) from referring to each other directly, thus reducing the number of connections. This pattern is similar to Façade in the sense that it provides a common interface for a group of classes to use. In the case of Mediator, however, there is no need to separate the classes into groups (although it is possible), but all classes can both require methods from the interface provided by the Mediator, as well as provide an interface for the Mediator. The Mediator, similarly to Façade, increases modifiability through decreasing coupling and increasing encapsulation. The Mediator is here considered from a class oriented viewpoint, and object related matters are omitted.

The Adapter is designed for situations where a class requires an interface, and a suitable interface already exists, but it is incompatible with the request. Thus, the Adapter provides a technical interface which is compatible with the request, and a technical class which then accesses the actual interface. Using the Adapter also makes it easy to change the final interface, if needed, as the requesting class is now always provided with the compliant Adapter interface.

The Strategy pattern aids in situations where there are several possibilities for implementing a certain algorithm (method). When alternatives need to be provided, Strategy provides a common interface which is implemented by different classes, all providing a somewhat different implementation of the same method. Again, as with Mediator, the Strategy is defined only in terms of changes to the class structure; and object-related matters (creating a concrete Strategy object, etc.) are not considered here.

The Template Method pattern is used when a method (the "template method") has both invariant and variant parts. The invariant parts can be implemented straightforwardly, but there should be an easy way to change the implementation of the varying parts. When applying the Template Method pattern, the class containing the "template method" is made abstract, and the varying parts of the "template method" are defined as abstract, primitive methods. These primitive methods are given concrete implementations in a subclass. The inherited methods now override the original ones, and if they need to be changed, only the subclass needs to be modified.

2.1.3 Software architecture quality metrics

There are several ways to perform software architecture evaluation, e.g., using simulations, mathematical modeling, experience-based assessment and scenario-based evaluation [Bosch, 2000]. In object-oriented design, software metrics are also a useful way to predict the quality of the design. The most relevant ways of evaluating the quality of software architecture

regarding this thesis are using software metrics and scenario-based evaluation, which are discussed in the sequel in more detail.

Metrics are usually based on some combination of calculating coupling between classes and cohesion within classes. High coupling and low cohesion indicate poor design choices in terms of modifiability, efficiency and maintainability. More refined metrics are also available (as given, e.g., by Losavio et al., [2004]), but they either require intricate information on the implementation of the software system, or a human expert is required for some portion of the evaluation process, i.e., not all values can be automatically deduced from the system. While metrics have the drawback of not being able to capture some essential entities of software systems which cannot be deduced from code or class structure, they have the benefit of being able to examine the entire system. Thus, given well-defined conditions, metrics can be used to evaluate the system in, theoretically, any given situation. Using human expertise, however, is always dependent on the knowledge and experience of the particular experts making the evaluation. Other experts, even using the same quality attributes as a basis for their evaluation, may produce a different result. Humans may also have difficulties in evaluating the system as a whole, if it is very large. However, as opposed to metrics, humans are not confined to structures and are able to grasp very abstract concepts, and can thus make very complex evaluations.

The most known metrics suite is presented by Chidamber and Kemerer [1994] (called hereafter the CK metrics), and it is based on four principles that rule object-oriented design process: identification of classes (and objects), identification of semantics of classes (and objects), identification of relationships between classes (and objects) and implementation of classes (and objects). Based on these principles, Chidamber and Kemerer [1994] formulate a metrics suite that consists of six different metrics: weighted methods per class (WMC), depth of inheritance tree (DIT), number of children (NOC), coupling between object classes (CBO), response for a class (RFC), and lack of cohesion in methods (LCOM). CBO and LCOM are classic indicators for modifiability, as discussed above. WMC indicates reusability. A high value for DIT predicts negative aspects of complexity and maintainability but a positive aspect of reusability. A high value of NOC indicates that extensive testing should be performed, but this metric can also be used to evaluate efficiency and reusability. Finally, RFC can be used to measure understandability, maintainability and testability.

A combination of the CK metrics or some modified versions of them are often used in maintenance work, e.g., in refactoring [Du Bois and Mens, 2003]. However, as the metrics are very basic, also the refactoring operations based solely on these metrics are usually simple, such as

moving a method or somehow modifying the class hierarchy. As the CK metrics are widely used and the basis for many other metric suites, their validity has also been studied by, e.g., Briand et al. [2000] and Olague et al. [2007]. The more recent study by Olague et al. [2007] concludes that the CK metrics, as well as the QMOOD (Quality Model for Object-Oriented Design) suite [Bansiya and Davis, 2002], which will be discussed in the following, are truly effective in detecting error-prone classes.

One example of metrics based on the CK metrics suite is given by Sahraoui et al. [2000]. They present a list of inheritance and coupling metrics, where the simplest metrics are NOC, CBO and number of methods (NOM), which is a simpler form of WMC. The rest, however, are more specialized extensions of the CK metrics, such as class-to-leaf depth, number of methods overridden and information-flow-based inheritance coupling. Detailed metrics such as these can be used to apply more sophisticated refactorings, such as creating specialized subclasses and aggregate classes.

As in the ISO standard [2007], the software architecture definition should also consider its evolution; no software system stays the same during its lifecycle, as changing requirements and maintenance operations alter the system. Mens and Demeyer [2001] present evolution metrics, which help evaluating how suitable the system is for evolution. Here the main metric is the distance between classes, which can, however, be defined as desired. The distance can measure, e.g., the number of methods, number of children or depth of inheritance tree. Large distances between classes can indicate a complex system.

Losavio et al. [2004] define the ISO quality standards model for software architectures. This model is somewhere in between pure metrics and evaluation using human expertise. The ISO 9126-1 quality model's characteristics are functionality, reliability, usability, efficiency, maintainability and portability [Losavio et al., 2004]. In the ISO model, the quality characteristics are refined into sub-characteristics, which are again refined to attributes. The attributes are then measured by metrics. Thus, the model needs human expertise in making the refinements, but the end result is a measurable value related to the architecture.

Finally, a more recent approach (compared to the CK metrics) to metrical software evaluation is the QMOOD metrics suite by Bansiya and Davis [2002]. Bansiya and Davis wanted metrics that would be particularly suitable for the design phase. Thus, they defined the quality attributes for QMOOD to be functionality, effectiveness, understandability, extendibility, reusability and flexibility. These attributes are evaluated by design properties, which can be refined into metrics. There are in total 11 different design properties, and one metric for each property. These

properties are then combined in different ways in order to evaluate the quality attributes. The actual metrics are design size in classes, number of hierarchies, average number of ancestors, number of polymorphic methods, class interface size, number of methods, data access metric, direct class coupling, cohesion among methods of a class, measure of aggregation and measure of functional abstraction. The QMOOD metrics suite is not that different from the CK metrics in the end, but does take some more intricate information about the design into account. The QMOOD metrics suite also defines what metrics to use for which quality attributes more clearly than the CK metrics, and the actual quality attributes are clear and well justified, which explains the popularity of the QMOOD suite.

To summarize, metrics are mostly elicited from class hierarchies and the different relationships between classes. Quality metrics are effective in indicating problems in object-oriented designs, particularly when considering efficiency, modifiability and complexity/understandability. Thus, metrics can be used as guidelines to determine the quality of object-oriented systems, but they usually require quite intricate information about the design, and some more sophisticated metrics require information that is not available before implementation. The usefulness and accurateness of different metrics suites have been demonstrated in several studies.

2.1.4 Software architecture evaluation

When moving on to evaluation made by humans, the evaluation usually begins with the following questions: How does an architecture resolve quality requirement x? Why is one solution better than another? These questions alone demonstrate the difference between using metrics to give values to quality attributes and using human expertise to evaluate the system: no metric can answer questions like "how" and "why" when discussing the positive and negative aspects of different architectural options. Metrics may also give very good scoring to individual quality requirements, but as a whole, the architecture may not be at all suitable for the system in question. Hence, although metrics can aid in architecture evaluation and are basically the only way of automated evaluation, they cannot completely replace the evaluation of experts.

The most widely used and known method for architecture evaluation is the Architecture Trade-off Analysis Method (ATAM) by Clements et al. [2002]. The main goals of ATAM are to define the key quality attribute requirements concerning the architecture, to refine design decisions for the architecture, and based on the two previous goals, to evaluate the

architectural design decisions to determine if they fulfill the quality attribute requirements satisfactorily.

Each quality attribute is evaluated with a set of *scenarios*. A scenario basically describes an interaction between a stakeholder and the system. Scenarios are used to analyze whether the architecture fulfills the critical quality requirements and to see potential risks involved in the architecture. A scenario describes a prospective real-life situation related to a particular quality attribute. For example, usability related scenarios mostly deal with the user interface and its implementation, while maintainability or modifiability related scenarios might try to predict changes or additions to functionalities or their implementations. Scenarios are given a level of priority and in the case of change scenarios, an estimate of their probability. Based on these, experts can concentrate on the most likely and most important scenarios. While the initial analysis regarding quality requirements is done by software architecture experts, all possible stakeholders are involved in brainstorming the scenarios in order to find all possible risks and considerations involved with the architecture [Clements et al., 2002]. After the scenarios have been assessed, the architecture should be evaluated against the scenarios, and changes to the design should be made accordingly, if the initial design does not meet the quality requirements satisfactorily.

As can be seen, ATAM relies purely on human expertise, and the evaluation of architecture is partially done simultaneously with development. Some basic architectural approaches are first presented based on the known structure of the system, and as the quality attribute requirements of the system become clearer, the architecture undergoes several iterations of analysis, while the architecture is being refined and different approaches may be considered. The "goodness" of the architecture can be defined and measured by how well it satisfies the quality requirements and how "easily" it responds to the scenarios related to the quality attributes.

To summarize, human-based evaluation relies on both knowledge of the system as well as guesswork concerning potential changes to the system and its environment. The main benefit of human-based design is the ability to capture all critical quality requirements, as there are no limits to how they can be expressed and measured. Thus, human evaluation usually gives a more dependable evaluation of the quality of an architecture than metrics, as abstract concepts are not detectable by metrics. Also, the collected expertise of stakeholders performing, e.g., the ATAM analysis, is far beyond what can be achieved with metric calculations.

2.2 GENETIC ALGORITHMS

Genetic algorithms belong to the family of meta-heuristic search algorithms, and are used as a way to seek for solutions in a very large search space. When a deterministic search is not feasible, but the problem can still be formalized as a search problem, genetic algorithms provide a sophisticated way to quickly search for a sufficiently good solution [Mitchell, 1996]. Holland [1975] structured the idea of GAs as a way for computer science to take advantage of the phenomena present in natural evolution.

Evolution is described by Darwin's theory of natural selection: the individuals that are best fit to the environment they are living in will survive, and the ones who are poorly adjusted to the environment will die. Over time, this kind of natural selection will ensure that species that have been able to produce properties that enhance their probability of survival will be spared. Thus, such species will be able to produce fitter offspring.

Genetic algorithms, by definition, work with the same concepts that are involved in the evolution of species. The core of each living being is in its DNA. DNA, in turn, can be separated into strings of *chromosomes*. Each chromosome consists of *genes*, which specify different properties for the individual. For example, one gene is in charge of the color of eyes, one of the shape of the face, and so on. Each gene usually has several options, e.g., the eyes can be blue or brown and the face can be round or oval; these different variations are called *alleles*. A gene always has a specific place in the chromosome; this is called the *locus.* Collectively, the chromosome(s) containing these genes specify the entire individual. A set of individuals at a certain time point is called a *population*, and *generation* indicates how many iterations the evolution has performed.

Naturally, for evolution to appear, there must be reproduction in order to bring new individuals into the population and thus enable natural selection. Reproduction is most commonly achieved with mating between two individuals, which is referred to as *crossover*. Crossover combines parts of two individuals (parents) and aims at producing offspring, which contain genes from both parents. The offspring is then added to the existing population. When the amount of individuals exceeds the amount a population can hold, natural selection occurs to discard the unfit individuals. Crossover is, however, very slow when an individual needs to adapt to a quickly changing environment. In order to accommodate to a new environment, *mutation* is required. On chromosome level, mutation usually changes the value of one or more genes (i.e., one allele is chosen over another, and new possible alleles may be produced).

In order to know which individuals are better fit than others, the individual's *fitness* is calculated. The fitness is related to the probability

that an individual has for participating in crossover and surviving through to the next generation.

A formalization of the GA is given in Algorithm 1. In the following subsections I will go over how to work with a genetic algorithm in more detail by using the knapsack problem as an example. In the knapsack problem, we have a set of items that all have different weights and volumes, and a subset of those items should be chosen so that their combined weight is maximized, while the combined volume cannot exceed the volume of the knapsack.

Algorithm 1 geneticAlgorithm

Input: formalization of solution, *initialSolution*

```
population ← createPopulation(initialSolution)
while NOT terminationCondition do
    foreach chromosome in population
        p ← randomProbability
        if p > mutationProbability then
        mutate(chromosome)
        end if
    end for
    foreach chromosome in population
        cp ← randomProbability
        if cp > crossoverProbability then
        addToParents(chromosome)
        end if
    end for
    foreach chromosome in parents
        father ←chromosome
        mother← selectNextChromosome(parents)
        offspring ← crossover(father, mother)
        addToPopulation(offspring)
        removeFromParents (father, mother)
    end for
    foreach chromosome in population
        calculatefitness(chromosome)
    end for
    selectNextPopulation()
end while
```

2.2.1 Encoding

The very first step when applying genetic algorithms is to encode individuals (the possible solutions) so the algorithm can work with them. As stated, the GA works with concepts from biology, so the encoding mechanism should be such that the solution can be thought of as a chromosome and be divided into genes. Apart from that, there are no restrictions regarding encoding.

The best encoding mechanism depends on the problem in question. What might work well for one problem might not be sufficient or even feasible to encode another type of problem. The most common encoding mechanism is to use a string or vector of bits, so that one bit corresponds to one gene. A bit in a specific index (locus) usually indicates the inclusion (bit value 1) or exclusion (bit value 0) of the data represented by that gene, i.e., the different variations 0 and 1 are considered alleles, specifying a certain property for a particular gene in one locus. However, this form of encoding is highly insufficient for more complex problems. For example, if the problem is an ordering problem (e.g., the traveling salesman problem, where the goal is to find the optimal order for visiting the cities), a simple inclusion/exclusion encoding is not enough, as all the bits (cities) need to be included. For the most complex problems, even a single value isn't enough, but the gene itself may also be a vector. The gene may also contain different kinds of fields for different types of data; in this case the gene may be called a *supergene* [Amoui et al., 2006].

If we consider the knapsack problem, the simple bit vector encoding is quite sufficient. For n items, the solution can thus be encoded as a vector with n bits, where 0 represents not including the corresponding item in the knapsack, and 1 represents including it. The order of the items can be anything, the most natural one being ascending based on either weight or volume (i.e., the item represented by the gene in locus 1 would have the smallest weight or volume). The only requirement is that each item is fixed to be represented by a specific bit. The information about the weights and volumes can be stored separately. Thus, if in our example we have seven items, and items four and seven are included in the knapsack, the individual representing this solution would be encoded as 0001001.

When the GA begins to search for a solution, an initial population is first needed. An initial population can be constructed roughly in three ways: 1) creating random individuals by administering mutations to a "null" individual, 2) creating the individuals intelligently, or 3) inserting "ready" individuals into the population. The best policy for creating the initial population depends on the problem at hand, and basically anything goes as long as the GA is provided with a starting point to begin the evolutionary process.

2.2.2 Mutations

After an initial population is created, mutations are applied for each individual in order to create diversity. The actual implementation of mutations is completely dependent on the encoding of the solution. However, a mutation most often changes the value of one gene. If the gene is a combination of values, i.e., is a vector itself or a supergene, a mutation

can change the gene completely or just one value within the gene. The problem can permit several mutations, i.e., an individual can be modified in many different ways. Whatever the mutations are, the result should always still be a feasible solution. It is also possible to have a separate "correction mutation" that will check the chromosome after a mutation to see that it is still valid. If the mutation has caused the chromosome to become unnatural, i.e., it does not belong to the solution space anymore, corrective actions will take place. Such actions do not necessarily just revert the mutation that caused the problem, but might do even bigger changes to the chromosome. The chromosome can even be completely discarded from the population.

The locus where the mutation is applied to is usually selected randomly, but in some cases intelligent methods can be used to search for particular genes where mutations should be performed. Mutations are also given a probability, the so-called *mutation rate*. If the mutation rate is 10%, then during one generation, approximately 10% of the individuals will be subjected to that particular mutation. Thus, not all individuals are mutated during every generation, as mutation probabilities are quite often rather low.

In the knapsack example, only one logical mutation exists: changing a bit from 0 to 1 or vice versa. Considering the solution in the previous subsection, we may now mutate the solution so that we apply the mutation to locus three. The bit vector would now change from 0001001 to 0011001, and item three is included in the knapsack. Figure 3 illustrates the mutation.

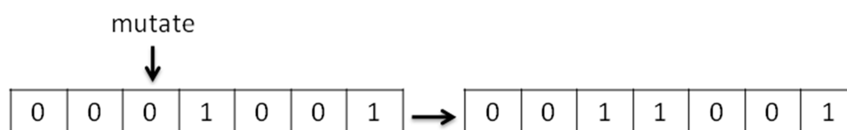


Figure 3. Mutation

2.2.3 Crossover

After mutations are administered to the chromosomes, the GA proceeds to perform crossover. Crossover combines parts of two individuals (the parents) in order to produce new solution(s) (their offspring) to the population. Crossover is also given a probability, i.e., a specified *crossover rate*, and is usually applied more often than mutations. This probability, i.e., the likelihood of an individual being involved with crossover, can at its simplest be randomly assigned (as in Algorithm 1), but in practice this is rare. It is more common to select parents based on their fitness values or ranks, so the better a fitness one has, the bigger is the likelihood of being

involved in crossover, and thus passing on favorable properties to its offspring.

The crossover operation can combine simply two "halves" of parents (one-point crossover) or several blocks of genes (two-point or multi-point crossover). Selecting the crossover point or points can be done randomly, or heuristics can be used to find the best crossover point(s) to preserve the most desired qualities of both parents. If the crossover points are selected randomly, it is common to produce at least two new individuals. If, however, the crossover points are selected in a purposeful way to combine the best parts of parents, it is more natural to produce only one offspring, as another(s) only contains "leftover" genes. The offspring can replace the parents or just be added in the generation. If they replace the parents, then it is assumed that offspring is always better than the parents, as there are no "extra" individuals that could be discarded in natural selection.

In the knapsack example, we can perform a simple single-point crossover where the crossover point *cp* is selected randomly. Crossover takes the genes (bits) from one individual (mother), from loci 1 to *cp,* and the bits in loci from *cp*+1 to *n* from another individual (father). These genes are combined to produce a new individual (child1), which now contains bits from both parents. Another child (child 2) can be straightforwardly created by taking genes from the father from loci 1 to *cp* and genes from the mother from loci *cp*+1 to *n*. An example of perfoming the crossover for the knapsack case is given in Figure 4. Here the crossover point is in locus three, and two children are produced.

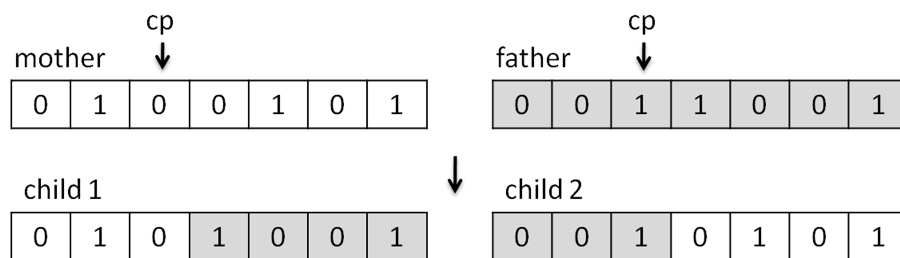


Figure 4. Crossover

2.2.4 Fitness

The fitness function determines the quality of a solution. For example, in the case of the travelling salesman problem the fitness would be the length of the route, and it should be minimized. However, in the case of creating a test case for a software system, the fitness would be the coverage of the test case, which should be maximized (and is far more complex to calculate than the length of the route for the salesman). Correctly defining the fitness function is crucial, as it is the only thing guiding the genetic

algorithm in its search for as good solutions as possible. The fitness function rarely gives an absolute quality of a solution, i.e., if one solution has a fitness value 1, and another a fitness value 10, it should not be assumed that the second solution is ten times better than the first one. Rather, the fitness value should be thought of as a relative indicator of the order of solutions, when examined against the quality attributes.

In the case of our knapsack example, the fitness function is simple: the fitness function is the sum of weights of the included items, while regarding the constraint that the volume cannot exceed a set limit. For our example with seven items, we can specify that item one has a weight of one unit, item two weighs two units, and so on, and their volume would be reversed, i.e., item one has a volume of seven units, item two has a volume of six units, and so on. We assume that the knapsack has a maximum volume of 10. The fitnesses of the individuals in Figure 4 can be determined as follows: the fitness for the mother is 2+5+7 = 14, for the father 3+4+7 = 14, the fitness for the first child is 2+4+7 = 13, and the fitness of the second child is 3+5+7 = 15. Thus, the best solution would appear to be the second child, with a fitness of 15. However, we need to consider the volume restriction. The volumes are 10 for the mother, 10 for the father, 11 for the first child and 9 for the second child. As the maximum volume is 10, the first child is discarded, and the second child remains the fittest individual.

2.2.5 Selection

As according to the biological analogy, natural selection is applied in the GA to discard the weakest individuals (with respect to the used fitness function) from the population. As crossover produces "extra" solutions, the size of the population at the end of one generation is always bigger than the specified population size. Selection can thus simply discard as many individuals as have been added through crossover, and so the population size will always be the same at the start of each generation. The selection operator should be defined so that the fittest survive (as in natural selection) but also so that there still remains variation in the population so the evolution does not get stuck to a local optimum.

The simplest way of defining a selection operator is to use a purely *elitist selection*. This selects only the very best individuals in terms of fitness. Elitist selection is easy to understand and simple to implement; one can simply discard the weakest individuals in the population. However, elitist selection on its own is not an ideal selection operator, as elitist algorithms tend to converge towards local optima too quickly.

Another way of defining the selection operator is to use a *fitness-proportionate* selection. In this case, the probability of being selected for the next generation depends on fitness values. However, differences in actual quality are rarely directly comparable to absolute fitness values. In order to take this into account, the fitness-proportionate selection can be *rank-based*. When the selection is based on rank, the individuals are ordered according to their fitness values. The probability of being selected to the next generation thus increases according to the rank – not according to absolute fitness values. The basic fitness-proportionate selection as well as its rank-based variation can be implemented, for example, with a "roulette-wheel" sampling [Mitchell, 1996; Michalewicz, 1992; Reeves, 1995]. Here, each individual is given a slice of the "wheel" that is in proportion to the "area" that its fitness has in the overall fitness of the population. In simple fitness-proportionate selection, the size of the slice is calculated directly based on the absolute fitness values, while in rank-based selection the size is proportionate to the rank, and thus the sizes increase in a linear fashion when compared to the order of the individuals. Either way, the individuals with higher fitnesses have a larger area in the wheel, and thus have a higher probability of getting selected. The wheel is then spun for as many times as there are individuals needed for the population. An alternative to the roulette wheel method is to use, e.g., the *tournament technique* to select the next generation [Miller and Goldberg, 1995].

A common selection operator is a crossing of the two methods presented above; the survival of the very fittest is guaranteed by choosing a few of the best individual(s) with elitist methods, while the rest of the population is selected with the probabilistic method in order to ensure variety within the population.

There are different approaches to using the selection operator. Mitchell [1996] and Reeves [1995] consider that the selection operator selects the individuals that are most likely to reproduce, i.e., become parents. Michalewicz [1992] uses the selection operator in order to find the fittest individuals for the next generation. Both approaches keep the same selection probabilities for all individuals during the entire selection process, i.e., an individual with a high fitness value may be selected to the next population more than once. Thus, there are no specific rules to how the selection operator should be defined, as long as the GA generally follows the guidelines of natural evolution.

2.2.6 Parameters

Correctly defining the different operations (mutations, crossover and fitness function) is vital in order to achieve satisfactory results. However,

as seen in Algorithm 1, there are also many other parameters regarding the GA that need to be defined and greatly affect the outcome. These parameters are the population size, number of generations (often used as the terminating condition) and the mutation and crossover probabilities.

Having a large enough population ensures variance within a generation, and enables a wide selection of different solutions at every stage of evolution. However, at a certain point the results start to converge, and a larger population always means more fitness evaluations and thus requires more computation time. Similarly, the more generations the algorithm is allowed to evolve for, the higher the chances are that it will be able to reach better results. However, again, letting an algorithm run for, say, 10 000 generations will most probably not be beneficial: if the operations and parameters have been chosen correctly, a reasonably good solution should have been found much earlier.

Mutation and crossover probabilities both affect how fast the population evolves. If the probabilities are too high, there is the risk that the implementation of genetic operations becomes random instead of guided. Vice versa, if the probabilities are too low there is the risk that the population will evolve too slowly, and no real diversity will exist.

Thus, all parameters should be carefully considered. There is no single method for finding the optimal or "correct" parameters, but the most common way is to simply perform trial-and-error iterations until results are satisfactory.

2.3 SEARCH-BASED SOFTWARE DESIGN

Search-based software engineering has applied meta-heuristic search algorithms to software engineering problems since the '70s [Clarke et al., 2003]. Problems in the area of testing, especially in automated generation of test cases, have been most studied [Harman et al., 2009]. However, the area most relevant to this thesis is the field of search-based software design (SBSD). In SBSD, search algorithms are used to tackle problems related to different stages of software design: assigning methods to classes, designing the architecture with patterns or styles, QoS-optimization on service-oriented architectures, and re-design. While subjects more related to re-design (i.e., maintenance or re-engineering), such as clustering (e.g., [Di Penta et al., 2005; Doval et al., 1999; Harman and Tratt, 2007; Seng et al., 2005]) and refactoring (e.g., [Du Bois and Mens, 2003; O'Keeffe and Ó Cinnéide, 2006, 2008; Seng et al., 2006; Quaum and Heckel, 2009]), have been studied for a couple of decades, now also problems regarding direct software design, such as solving the class responsibility assignment (CRA) problem and applying design patterns, have attracted more interest. The most prominent studies in SBSD published (and electronically available)

by December 2009 are described in detail in publication [VI]. More recent studies are briefly discussed here.

2.3.1 Class design

Simons et al. [2010] study using evolutionary, multi-objective search and software agents to aid the software architect in class design. Use cases are used as a starting point. Actions (verbs) in the use cases are transformed into methods, and data (nouns) is transformed to attributes. The methods and attributes are then grouped into classes. Each class should have at least one method and one attribute, and no method or attribute can be in more than one class. One individual (solution) is thus the design containing all methods and attributes (and their class distribution). For mutation, a set of methods and/or attributes is selected and relocated to a different class. Crossover is applied so that the attributes and methods of two classes are swapped. Coupling and cohesion are used to calculate fitness.

The evolutionary search is performed by a software agent. Simons et al. suggest that a global multi-objective search is unnecessary, and the search should be narrowed towards the "most useful and interesting candidate designs". They attempt to achieve this by isolating discrete zones from the search space, and then using a local search within these zones. Local search is conducted using a single-objective genetic algorithm, which only considers coupling in the fitness calculations. The designer is then presented with the results of these local searches. The designer can manually specify diversity thresholds to control when a zone is isolated, or an agent can do it automatically.

Bowman et al. [2010] have applied a multi-objective genetic algorithm (MOGA) to assist in the CRA problem. The CRA problem is similar to the class design problem studied by Simons et al., as it attempts to solve the assignment of responsibilities (methods) into classes and the interaction between classes. Bowman et al. also use coupling and cohesion to measure the fitness of their solution, and aim rather at providing interactive feedback to a designer than at producing a whole design. Their MOGA takes a class diagram as input, as well as user-defined constraints on what can and cannot change in the class diagram. The class diagram is then evaluated, and possible improvements are suggested. The MOGA ultimately provides alternative solutions to the user.

2.3.2 Low-level architecture transformations

Jensen and Cheng [2010] present an approach based on genetic programming (GP) for generating refactoring strategies that introduce

design patterns. The authors have implemented a tool, REMODEL, which takes as input a UML class diagram representing the system under design. The system is refactored by applying "mini-transformations": abstraction, abstract access, partial abstraction, delegation, encapsulating construction, and wrapping. The encoding is made in tree form (suitable for GP), where each node is a transformation. A sequence of mini-transformations can produce a design pattern. A subset of the patterns specified by Gamma et al. [1995] is used: Abstract factory, Adapter, Bridge, Decorator, Prototype and Proxy. Mutations are applied by simply changing one node (transformation), and crossover is applied as exchanging sub-trees.

The QMOOD [Bansiya and Davis, 2002] metrics suite is used for fitness calculations. In addition to the QMOOD metrics, the authors also give a penalty based on the number of used mini-transformations and reward the existence of (any) design patterns. The output consists of a refactored software design as well as the set of steps to transform the original design into the refactored design. This way the refactoring can be done either automatically or manually; this decision is left for the software engineer.

2.3.3 High-level architectural transformations

Praditwong et al. [2011] introduce a multi-objective approach for automated software module clustering, as well as two formulations: the equal-size cluster approach and the maximizing cluster approach. The equal-size cluster approach favors clusters that have on average the same number of modules while the maximizing cluster approach favors a minimal amount of clusters with only one module. A two-archive Pareto optimal GA is mainly used, but a single objective hill climbing algorithm is also implemented for comparison. Their primary findings suggest that in order to have solutions with high cohesion and low coupling, the equal-size cluster approach to the multi-objective problem produces the best results overall. This study is an example of the several studies in clustering, which ultimately work more on the refactoring area.

Martens et al. [2010] present an approach which attempts to automatically improve a given architecture model with respect to performance, reliability, and cost. The approach is best suited for component-based software architectures. Martens et al. they attempt to optimize four degrees of freedom: processor speed, number of servers, component allocation and component selection. It is assumed that components with the same interface provide the same functionality, and thus no attention to the functionality of the system is needed. The approach is implemented in the PerOpteryx tool. The tool requires as input a component-based architecture model with performance, reliability and cost annotations. The tool then searches for Pareto optimal candidate solutions. When mutating,

one or several design options (degrees of freedom) are varied. In crossover some of each candidate's design option values are taken and combined. Solutions that violate quality requirements are eliminated during selection. After elimination, only the Pareto optimal solutions are kept. The authors list a significant amount of limitations to their approach, such as questionable efficiency, no regard for uncertainties, limited degrees of freedom, simplistic cost model and limited genetic encoding (an array of choices).

Aleti et al. [2009] present the ArcheOpterix tool. It is an Eclipse plug-in that provides a platform to implement different architecture evaluation and optimization algorithms, and uses AADL as the architecture description language. So far, only deployment metrics (data transmission reliability and communication overhead) have been implemented for evaluation, and the authors present an example of optimizing a deployment architecture. ArcheOpterix uses Pareto optimality as a primary method for finding best solutions, but a single-objective fitness function is also implemented. When a near Pareto front has been found, ArcheOpterix draws the near Pareto front line, which contains all the non-dominated solutions found by the algorithm.

2.3.4 Comparison to presented work

The studies by Simons et al. [2010] and Bowman et al. [2010] are close to the approach presented here, as in the case of Simons et al., the direction of design is clearly upstream, and with Bowman et al. there is also potential for upstream design. However, both stay at class level, and do not consider how the actual classes interact. Also, both only use coupling and cohesion as metrics, which is natural when dealing with "simple" class structure only, but insufficient if, e.g., interfaces are considered, as their value is not apparent if the fitness relies solely on these simple metrics.

While Bowman et al. [2010] and Simons et al. [2010] study lower level design than what is discussed in this thesis, the studies by Praditwong et al., [2011], Martens et al. [2010] and Aleti et al. [2010] operate on a significantly higher level. These studies assume that the class level design is complete and concentrate on components. Martens et al. [2010] even assume that they assume that interchangeable components produce similar functionality, which is a fair assumption given that they attempt to optimize the architecture in terms of performance, reliability and cost, and the main methods of doing this is reallocating components to servers and considering the number of servers used. However, this assumption makes it clear that the designs of the actual components are not under consideration. As for the study by Aleti et al. [2010], they have

implemented a tool which should be capable of using different evolutionary approaches, fitness functions and architectures, but only provide an example for a deployment architecture. So, at this point it seems that they are not interested in the class level design of the architecture. Finally, the study by Praditwong et al. [2011] is an example of the many studies made in software clustering. It differs from the approach presented in this thesis both on the direction of the design and on level of detail, as studies in clustering rarely consider what the modules contain and how the clusters are ultimately connected, but are only interested whether they are connected or not.

The approach by Jensen and Cheng [2010] is the most similar to the one presented in this thesis. However, they clearly have a refactoring oriented approach instead of pure upstream design. They also construct design patterns piece by piece through mini-transformations, instead of applying whole patterns. The existence of patterns, no matter what kind, is, however, greatly rewarded. Obviously, there is a risk that a large number of incomplete patterns is left in the architecture, and that the patterns are not applied in the best places.

To summarize, the more recent approaches give a fairly good view of the overall status of search-based software engineering. Upstream approaches are still scarce, and the present studies only change the design in very small steps. However, when a good architecture "simply" needs to be optimized, there are many studies on different levels on how to do that automatically. Using design patterns is one of the most recent trends, but no other study applies them to upstream software architecture synthesis in a manner similar to the one used in this thesis.

3 Genetic Software Architecture Synthesis

In this chapter the actual method for genetic software architecture synthesis is described, and implementing the synthesizer is discussed. Experimental results with this approach have been presented in publications [I], [II] and [VIII]. Two case studies were used in the experiments: the ehome, and a robot war game simulator, called hereafter robo. The approach is, however, applicable for any system in principle. I will here describe the method on a general level, and use ehome as a running example to demonstrate how the synthesis process works in practice. Results from case studies are briefly presented. Also, I will briefly present the Darwin tool, which provides a user interface for software architecture synthesis with the presented algorithm. The tool support is discussed in more detail in publication [V].

3.1 METHOD

Genetic software architecture synthesis, by definition, uses GAs to synthesize software architecture design. Thus, the requirements set by GAs themselves, as defined in Section 2.2, dictate how the synthesis is actually implemented. In this section I will describe how functional requirements are encoded for the GA, how the architecture is transformed through mutations and the search space explored with crossover, and how the different quality attributes are expressed in the fitness function. The result of the synthesis process (the best individual in the last generation) is illustrated as a class diagram. The GA described here has been implemented with Java 1.5 as a command-line based program. The

original synthesizer implementation uses UMLGraph [2011] and GraphViz [2011] for producing the class diagrams.

3.1.1 Input

Each software system is designed to serve a certain purpose. This purpose is formulated in functional requirements. As GAs cannot synthesize the actual purpose of the software system, the specification of functional requirements must be done manually. However, only the minimal amount of effort should go into elaborating the requirements, as the idea is to exploit the power of the GA as much as possible. The functional requirements of the system under design are gathered in the way described in Section 2.1. To give a complete view of the synthesis process, I will here use a new example for ehome.

Specifying requirements begins with giving use cases. Here, I will take as an example the "adjust room temperature" use case for ehome. The use case is illustrated in Figure 5. The user simply places a command that the temperature should be adjusted (for the sake of simplicity, we can here consider elevation), and ehome adjusts the temperature by turning on the heater.

The sequence diagram for the temperature adjustment use case is given in Figure 6. The process begins with a call from the user to set the temperature to a new level. The system then calls the temperature regulation component, which measures the current temperature, and then sets the heater on. After the correct temperature is reached, the heater is turned off.

While sequence diagrams already give a good understanding of how the different operations depend on each other, a structural view still needs to be obtained, as patterns cannot be inserted into sequences of calls. Fortunately, sequence diagrams can easily be turned into *class diagrams*. At this point, the class diagram would not consist of anything but the classes, their methods and attributes, and connections between classes, as defined in the sequence diagram.

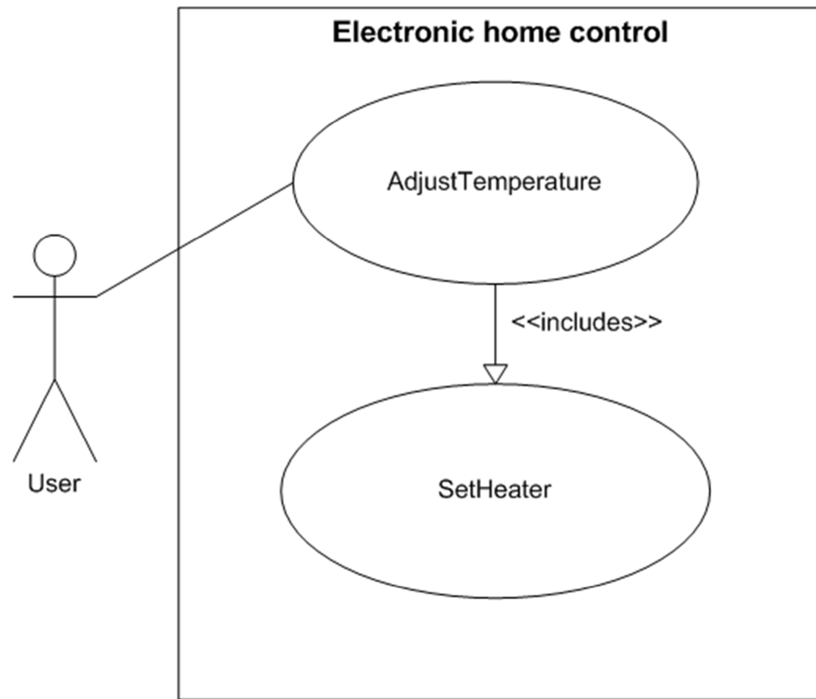


Figure 5. Use case for ehome (adjust temperature)

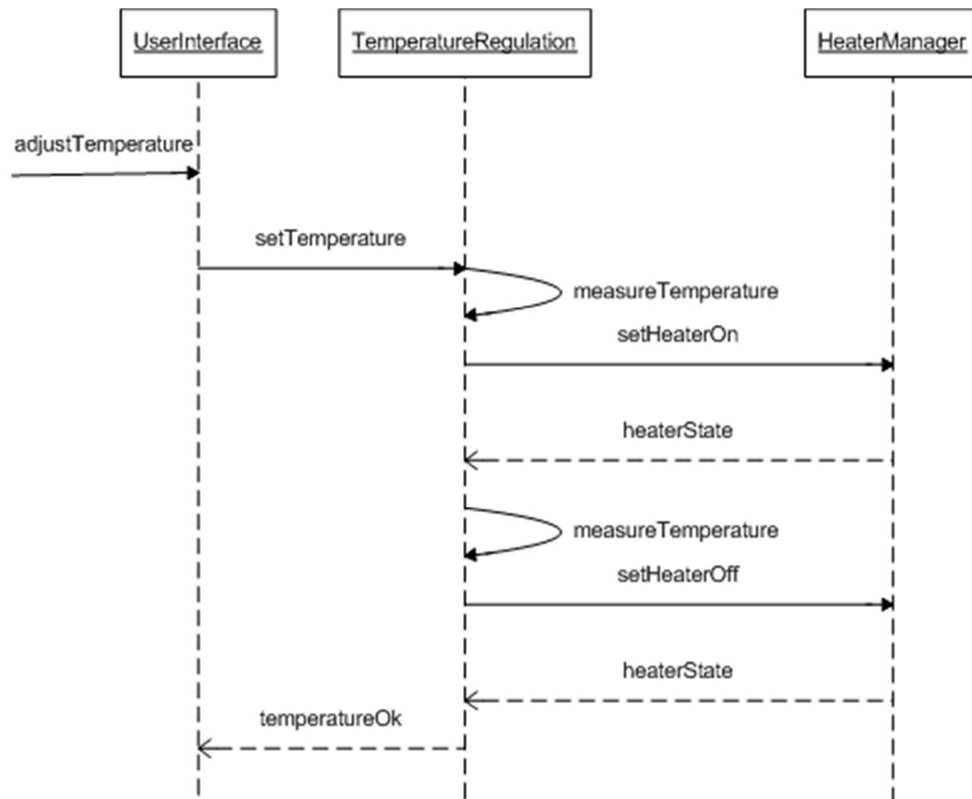


Figure 6. Sequence diagram for adjust temperature use case

Transforming a simplistic call sequence to a class diagram is straightforward. Using the temperature adjustment use case as an example, the class diagram would have three classes, the UI, the Temperature Regulation and the Heater Manager. It should be noted that

the "UI" considers here all user interface related matters, and handles all calls to the actual functional components. The UI may, in practice, contain several classes for handling all the user interface requirements. However, these implementation details are omitted here.

Calls between the different objects in the sequence diagram would be made into methods inside these classes. Information relating to temperature state would be an attribute in the Temperature Regulation class, and the class would contain accessor methods for it. The Heater Manager class would similarly have attributes for its state. In the class diagram, there would be "use" relations between the UI and Temperature Regulation, and between Temperature Regulation and Heater Manager classes (as according to the sequence diagram). The example class diagram for this use case is given in Figure 7.

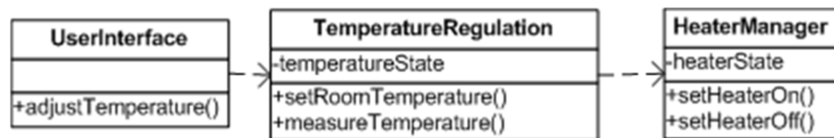


Figure 7. Class diagram example for ehome (temperature control)

Collecting all use cases would ultimately produce a complete class diagram of the system. This complete class diagram is called here the *null architecture*. As described, the null architecture is extremely simplistic. It does not contain any information about implementation or any specific design choices. The null architecture is merely a picture which gives a certain view of the gathered functional requirements.

The null architecture for ehome is given in Figure 8. Although here only classes are given, each class also provides an interface which is implemented by all operations (in that class) that are required by other classes. These interfaces can be seen as part of good design/coding practice, and not a part of actual design. Similarly, while it is assumed that attributes have accessor (getX() and setX()) methods, they are not portrayed in the null architecture, as they are considered as part of the coding process and standard good practice.

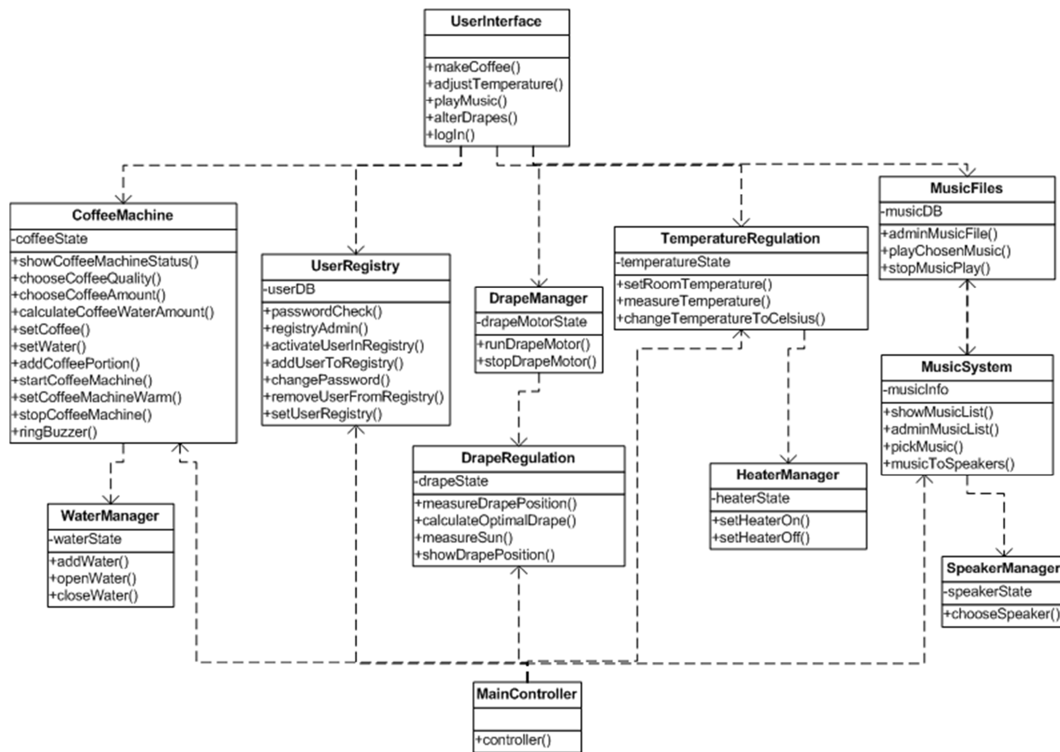


Figure 8. Null architecture for ehome

It should also be noted that while the calls given in the sequence diagram are marked as operations (methods) in the resulting class diagram, this does not dictate the actual implementation. In practice, each operation can, and often will, be implemented as a set of more detailed methods. The operations as used here describe functional entities (responsibilities) of the system.

Two case studies are used in this thesis, one of them being ehome, which has been used as a running example. The other case study is robo, which has been briefly discussed in Subsection 1.4. The null architecture for robo can be elicited through use cases and sequence diagrams (similarly to ehome). These steps will not be repeated here, but in order to compare the resulting architectures produced by the GA to the starting point, the null architecture for robo is given in Figure 9.

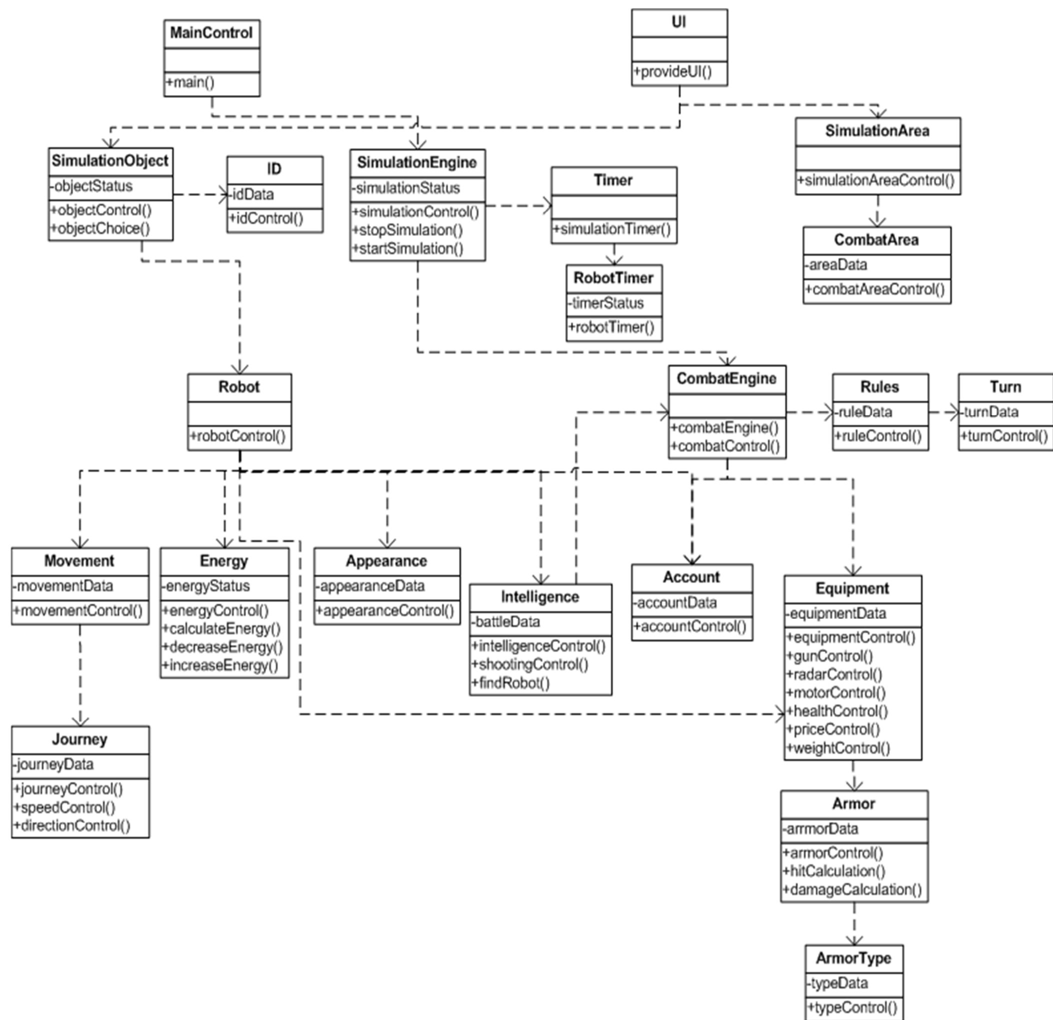


Figure 9. Null architecture for robo

In addition to simply specifying the functional requirements as a set of operations and dependencies between them, some characteristics of the operations can also be evaluated. For example, in the case of ehome, it can quite easily be deduced that calculating an optimal positioning for drapes requires much more data than, say, turning on the heater. Similarly, music is probably played much more often than a password needs to be changed. From the software development point of view, it is not difficult to predict that it is quite probable that several different options can be created for showing the music list or the coffee machine status. On the other hand, water will probably always be added to the coffee machine in a similar manner, and there will not be a need to provide optional implementations for that. Doing similar comparisons between all operations, relative values for parameter size, frequency of use and sensitiveness to variation (called hereafter simply variability) can be estimated. These, in turn, aid in achieving a more accurate evaluation of the architecture’s quality.

3.1.2 Encoding

The information given in the null architecture must be encoded for the GA to follow the biological analogies. There are two kinds of data regarding each operation o_i: basic information (given as input) and architectural information. The basic input information contains the operations $O_i = \{o_{i1}, o_{i2}, \ldots, o_{ik}\}$ depending on o_i, as can be deduced from the sequence diagram. An operation o_k depends on o_i if o_i is preceded by o_k in the sequence diagram. Furthermore, the basic information contains the name n_i, type d_i, frequency f_i, parameter size p_i, variability v_i and the predetermined null architecture class MC_i of the operation o_i. Naturally, the null architecture class is also architectural information, but it is specified in the input and not modified by the GA. Frequency, parameter size and variability can be given relative values: 1 for low, 3 for medium and 5 for high.

The architectural information, in turn, contains data regarding operation o_i's "place" in the architecture: the class C_i it belongs to, the interface I_i it implements, the message dispatcher D_i it uses, the operations $OD_i \subseteq O_i$ that call it through the dispatcher, and the design pattern P_i it is a part of. The dispatcher is given a separate field as opposed to other patterns for practical reasons. In practice, in the current implementation there is only one possible dispatcher that can be used, but the encoding leaves an opportunity to easily extend the approach so that several dispatchers may be present in the system.

All the information defined above is gathered together into one supergene, where each data particle is given a separate field. Thus, one supergene represents one operation in the system. Figure 10 depicts a supergene sg_i. If n is the number of operations in a system, the collection $<sg_1, sg_2, \ldots, sg_n>$ of these operations defines the entire system when collected into one chromosome (the chromosome thus being a vector of supergenes).

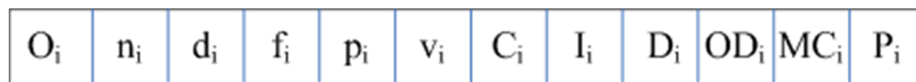


Figure 10. Supergene sg_i

Again, I use ehome as an example. In the null architecture phase, if the TemperatureRegulation class is given #ID 2 (and the interface #ID 2), the supergene for the operation "measureTemperature" (#id 9) would have the following values: O_9 = #idSetRoomTemperature, n_9 = measureTemperature, d_9 = f (as in functional), $p_9 = 3$, $f_9 = 1$, $v_9 = 3$, $C_9 = 2$, $I_9 = 0$, $D_9 = 0$, $OD_9 = 0$, $MC_9 = 4$, $P_9 = 0$. The interface has value 0, as measureTemperature is only required by setRoomTemperature, which is in the same class, and thus does need an interface to access this operation. The fields for message dispatcher and pattern have values 0, as no architectural solutions are included in the null architecture. As the

operation is here located in the original null architecture class, the values for C and MC are the same.

Note, that the encoding is indeed operation-centered. Thus, modifications to the architecture are considered from the viewpoint of how one particular operation can be accessed, and not how two classes communicate with each other. In practice, the null architecture is encoded into a text file, which is given as input for the algorithm, with each operation in its own line.

After encoding is done, the genetic synthesis can begin. However, a single base solution is not enough, as the GA requires a whole population to work with. The initial population is created by cloning the "base chromosome", i.e., null architecture, a desired number of times (in this work, 100 is used as the default), and then mutating each of the clones once. Thus, the initial population does not only contain clones, but has some versatility.

3.1.3 Mutation and crossover

Mutating the solutions (chromosomes) is done by inserting or deleting design patterns and software architecture styles. In addition, a "null mutation" is used, i.e., the chromosome remains the same for the next generation. The patterns and styles are those specified in Subsection 2.1.2. Each mutation is by default targeted to one operation (supergene) only (the message dispatcher makes a small exception, which will be discussed later). However, if the mutation (pattern) in question requires so, other supergenes may also be mutated in the process (as in the case of, e.g, Façade). Mutations are implemented as pairs of adding or removing a pattern. Each mutation (both addition and removal) is given a separate mutation probability. When adding a pattern, it is first checked that the operation is not already involved in another pattern, i.e., the field P_i must have value 0. Thus, the removal of patterns is only possible through the removal mutation – patterns cannot simply replace other existing patterns. Each mutation is now described from two viewpoints: how it is encoded in the supergene and how it affects the architecture.

Strategy: The precondition for the Strategy pattern is that the operation o_i, to which the pattern is targeted, is called by some other operation o_k from within the same class. When adding a Strategy pattern, a respective pattern instance SP is created. SP contains information of the common interface SI provided by the Strategy, the concrete implementing class(es) SC (as the actual number cannot be known at this point, only one class is used, which is enough to demonstrate the use of Strategy), and the operations which it concerns. The supergene sg_i (representing o_i) is then

updated so that the value for C_i is set to SC, the value for I_i is set to SI, and the value for P_i is set to SP.

When removing the pattern, the value for C_i is reset to the same value as MC_i (i.e., the operation is moved back to its null architecture class), the value for I_i is set to 0 and the pattern instance SP_i is removed, so the value for P is set to 0.

Figure 11 depicts how the architecture is altered when a Strategy pattern is added; again, ehome is used as an example, and the measureTemperature operation is subjected to mutation. Following the notation above, SC is represented by StrategyClass, SI by StrategyInterface, class C_i by TemperatureRegulation, operation o_i is measureTemperature and operation o_k is setRoomTemperature. Thus, SP would contain all this information (SC, SI, C_i, o_i and o_k), which defines the pattern. Here the Strategy class uses the original class, as the measureRoomTemperature operation requires temperature data from the TemperatureRegulation class. The TemperatureRegulation class may have an interface, but it is irrelevant here, as the data accessory methods are not provided by the interface (by default, each attribute is only used by operations in the same null architecture class).

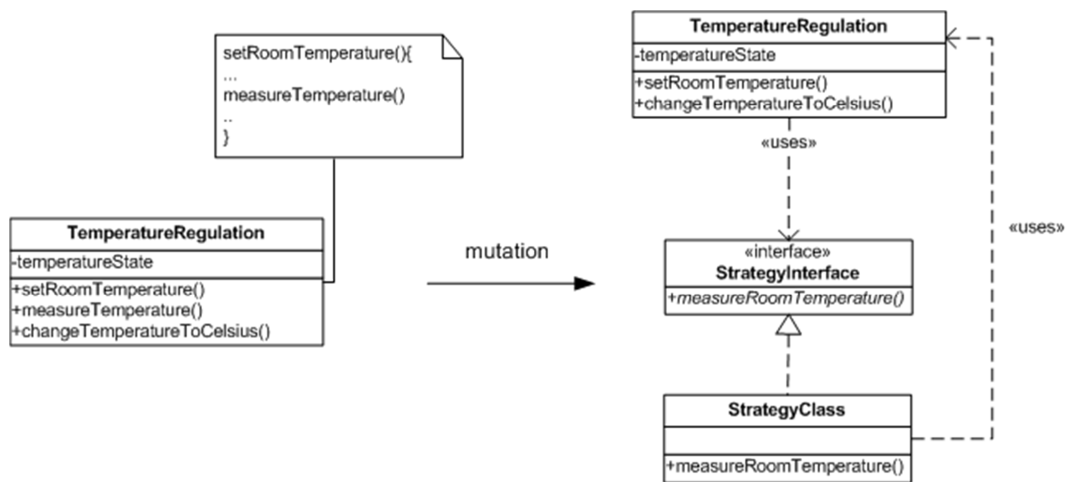


Figure 11. Strategy mutation

Adapter: The precondition for Adapter is simply that the operation o_i where it is applied is called by some other operation from a different class. When adding an Adapter pattern, a respective pattern instance AP is created. AP contains information of the common interface AI provided by the Adapter, the concrete implementing class AC, and the operations which it concerns. The supergene is then updated so that the value for P_i is set to AP. Removing an Adapter is done by simply resetting the value for P_i to 0.

The Adapter is illustrated with a technical class and interface, i.e., the adapter class and adapter interface do not contain specific methods, but simply illustrate where the Adapter should be applied. Figure 12 depicts how the architecture is altered when an Adapter pattern is added to access the operations in HeaterManager. Here, AC would be AdapterClass, AI is represented by AdapterInterface, and operations conserned with the pattern would be setRoomTemperature, which uses the HeaterManager, and setHeaterOn and setHeaterOff, which are the operations required by setRoomTemperature. AP would thus contain information of the aforementioned interface, two classes and three operations.

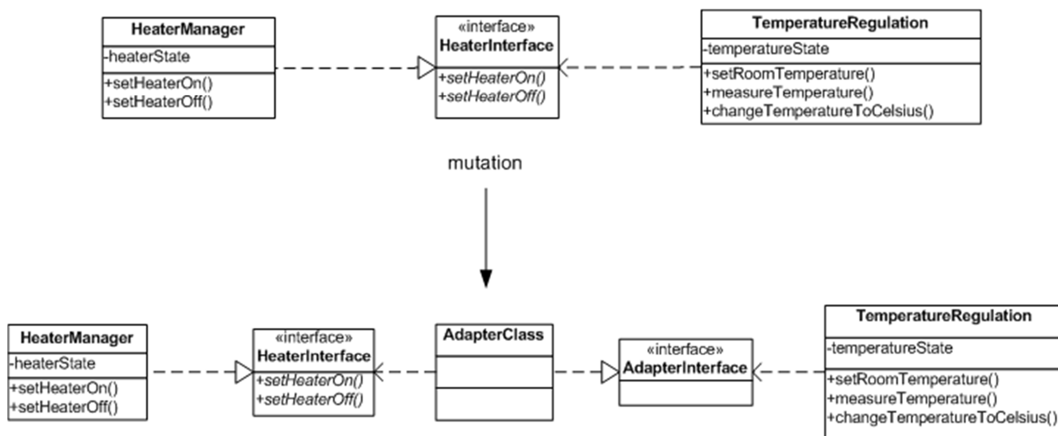


Figure 12. Adapter mutation

Template Method: The precondition for Template Method is that the operation o_i (the "template method"), where the pattern is applied to, must require (an)other operation(s) $o_k,\ldots,o_t$ from within the same class. When adding a Template Method pattern, a respective pattern instance TP is created. TP contains information of the concrete implementing (sub)class TC, and the operation(s) which it concerns. The supergenes of all operations o_i and $o_k,\ldots,o_t$ involved in the TemplateMethod are then updated so that the values for P_i and $P_k,\ldots,P_t$ are set to TP. Note, that the classes of operations are not changed, as the operations $o_k,\ldots,o_t$ in the TemplateMethod subclass still have abstract versions in the original class. The algorithm, however, knows the presence of the TemplateMethod by checking the Pattern field of the supergene, thus enabling evaluation based on it. Removing the pattern is done by simply changing the values of P_i and $P_k,\ldots,P_t$ to 0.

Template Method is illustrated with the subclass (containing the operations $o_k,\ldots,o_t$) which inherits the class where o_i resides. Figure 13 depicts how the architecture is altered when a Template Method pattern is added for setRoomTemperature. Following the notation above, here setRoomTemperature is the actual "template method" o_i, and measureRoomTemperature, being the required method o_k from within the same class, is placed in the subclass. In the example, the concrete subclass

TC is represented by TemplateMethodClass and the operations concerned are setRoomTemperature (the template method), and measureRoomTemperature (the interchangeable method).

In addition to creating a new subclass, the original class is made abstract. This, however, is not recorded in the supergene, as the fitness function does not consider abstract classes. The information of making the class abstract is encoded in the Pattern instance, and is only used for drawing the class diagram.

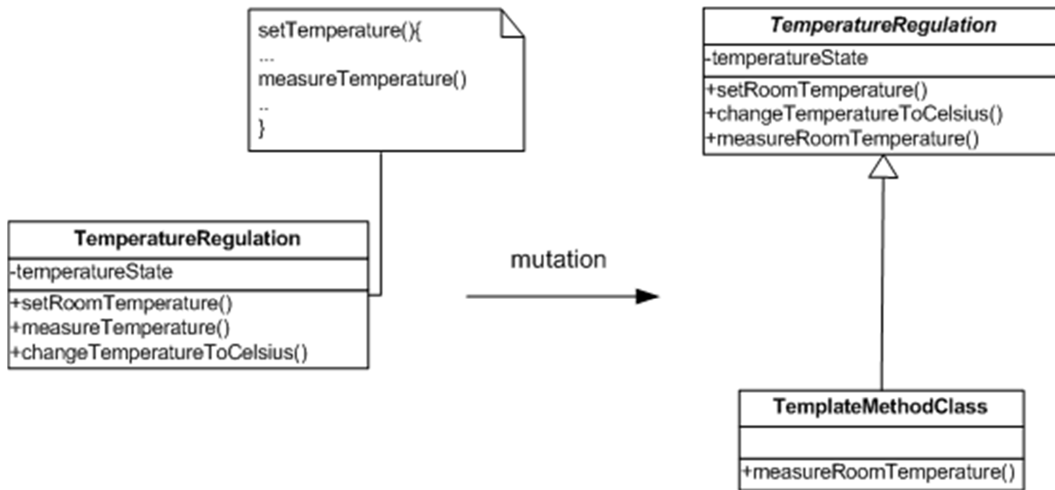


Figure 13. Template Method mutation

Façade: The precondition checks that a suitable structure can be found for the Façade, i.e., there must be a subsystem that could benefit from a common interface, as described in Subsection 2.1.2. When adding a Façade pattern, a respective pattern instance FP is created. FP contains information of the common interface FI provided by the Façade, the concrete implementing class FC, and the operations which it concerns. The supergene is then updated so that the value for $P_k,\ldots,P_t$ for each operation that is called through the Façade is set to FP. Removing Façade is done by simply changing the values for $P_k,\ldots,P_t$ for each involved operation to 0.

The Façade pattern is illustrated, similarly to the Adapter pattern, with a technical class and interface. Figure 14 depicts how the architecture is altered when a Façade pattern is added. Ehome's null architecture does not provide a natural placement for Façade, thus a general example is used here. The classes A,...,F can also have interfaces, but they are not portrayed here. Operations are also omitted, as on class diagram level, only connections between classes are shown. The pattern instance FP would in this example contain references to FacadeClass, FacadeInterface, and all the operations in classes A,…,F which are involved with the Façade pattern, i.e., either call an operation which is available through the FacadeInterface, or implement the FacadeInterface.

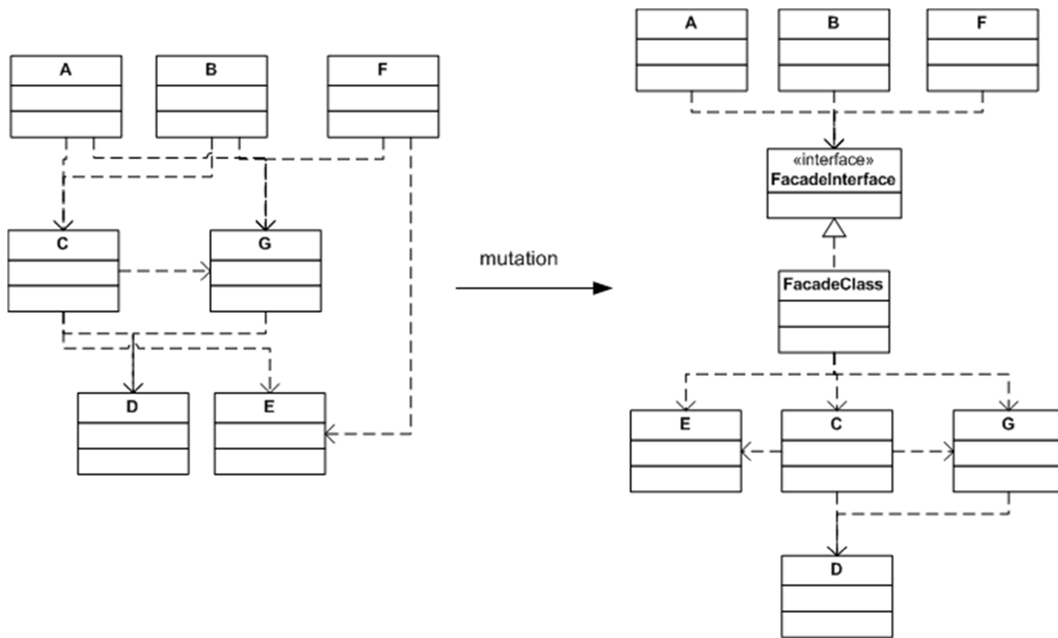


Figure 14. Façade mutation

Mediator: Adding and removing the Mediator pattern is done similarly to Façade (in terms of how the supergene is updated). The precondition is as described in Subsection 2.1.2; there must be a group of classes which communicate with each other. Mediator is illustrated in a similar manner as Façade. Figure 15 depicts how the architecture is altered when a Mediator pattern is added. Again, ehome's null architecture does not provide a natural placement for Mediator, thus a general example is used here as well. Similarly, interfaces and operations are omitted from the illustrations, and the pattern instance for Mediator is constructed in the same fashion as in the case of Façade.

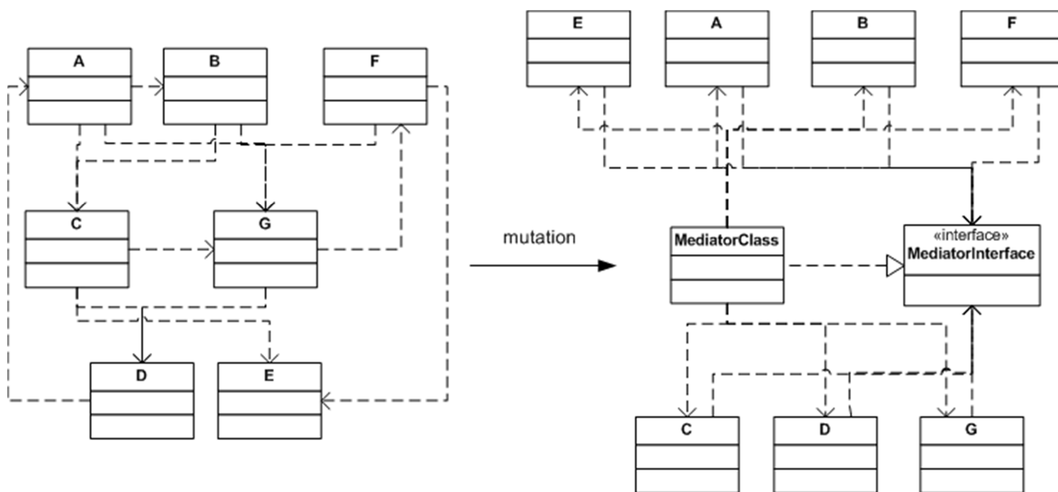


Figure 15. Mediator mutation

Client-server architecture style: Adding the client-server architecture style is made simply by creating a respective pattern instance, which is given in the pattern field of the chosen operation. The server is considered to affect

the entire class where the operation is. Moreover, there is a precondition that the class should contain at least three operations for it to be sensible to use a server connection to access them. Using the client-server style is illustrated with a "server" stereotype. Thus, any class using this server class becomes a client. Removing the pattern is done by simply setting the pattern value of the operations involved to 0. Figure 16 depicts how the architecture is altered when a server is introduced to one of the operations in TemperatureRegulation. Thus, the TemperatureRegulation class is stereotyped as server, and all classes using TemperatureRegulation become clients (not illustrated in the class diagrams).

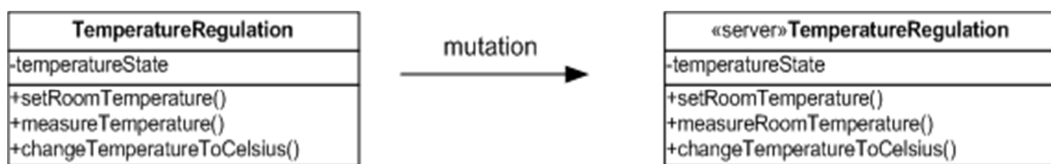


Figure 16. Client-server mutation

Message dispatcher architecture style: The message dispatcher architecture style is exceptional, as it must be introduced in two phases. In phase one the actual message dispatcher component is brought to the architecture. This is done by adding a "dummy" supergene to the chromosome. This dummy gene only contains information that the message dispatcher is present (D = 1), and all other values are 0. After the presence of a message dispatcher is verified, operations can use the message dispatcher for communication (phase two).

Only connections between operations in different null architecture classes can be handled by the message dispatcher, i.e., if an operation is "temporarily" in a different class (for example, after applying a Strategy pattern), it cannot use the message dispatcher to access, e.g., the data, from its original null architecture class. Adding a message-based communication to operation o_i is done by simply adding one of the depending operations (i.e., one of the operations in set O_i) to the set of operations which is using the message dispatcher to communicate with o_i (set OD_i). If operation o_i does not receive any direct calls but only messages from the dispatcher, then $O_i = OD_i$.

Removing a message dispatcher connection to o_i is done by simply removing the respective connecting operation from the set OD_i. Removing the entire message dispatcher is more difficult; the message dispatcher can only be removed if no operation is using it for communication. Thus, in order to completely remove the message dispatcher style, each connection between operations must be removed from the message dispatcher, after which the dispatcher itself can be removed (by deleting the dummy gene).

Figure 17 depicts how the architecture is altered when two operations use message dispatcher for communication; in particularly, the setHeaterOn

operation is subjected to mutation. It is assumed here that the message dispatcher is available in the architecture. In the example, setHeaterOn is required by setRoomTemperature, and thus the (ID of the) latter is a part of the dependingOperations set O for the former. After the mutation, the ID for setRoomTemperature operation is added to the set OD of setHeaterOn.

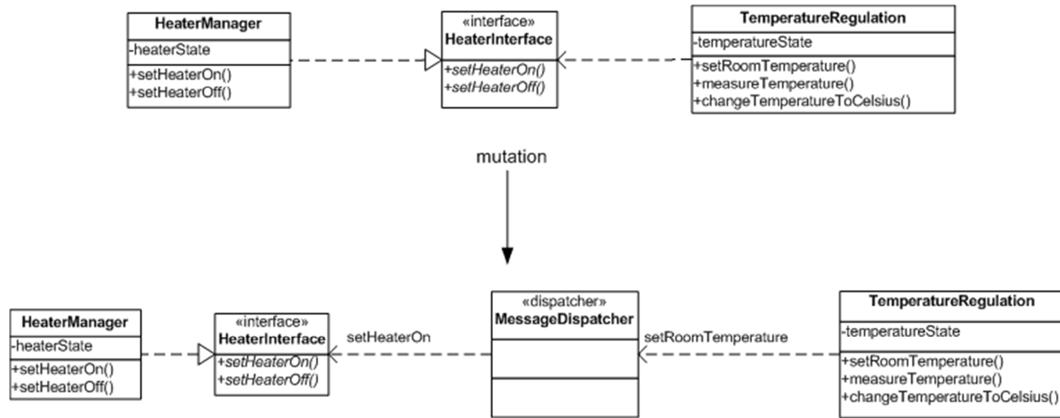


Figure 17. Message dispatcher connection mutation

Although the presented approach uses only the defined small set of object-oriented design patterns and architecture styles as mutations (transformations), it should be noted that it is by no means limited to such patterns. Any kind of transformation to the architecture can be used as a mutation. For example, simply changing the class of an operation could be used as a mutation. Also, new patterns suitable for a certain system under design can be defined for the synthesis, as transformations basically operate by simply creating new classes and interfaces and reallocating operations – the GA does not know whether this collection of changes to architecture actually corresponds to some known pattern.

The crossover is implemented as a random single-point crossover (see Subsection 2.2.3). Thus, one crossover point is selected randomly, and the crossover produces two new individuals. Both children are added to the population. In this approach, both parents are also kept in the population. The crossover is given a base probability. However, in this approach, the fitness of an individual affects its likelihood to participate in the crossover. If the individual is ranked in the better half of the population, its crossover probability is increased in linear proportion to its rank. However, if an individual is ranked in the lower half, its crossover probability is cut in half. This should aid better individuals to produce offspring more often than worse individuals.

The mutation probabilities and the crossover probability are collected to a “roulette wheel”, where the sizes of the slices of the wheel correspond to the probabilities. The actual probabilities can be set as desired. Here, the probabilities are given so that addition mutations have a bigger

probability than removal mutations, and the low-level patterns and architecture styles have bigger probabilities than medium-level patterns.

The "wheel" is then spun, to determine which mutation is performed. If the wheel lands on a mutation (not crossover), the wheel is spun a second time after the mutation is performed, as all individuals should have the possibility for mating in each generation, even if they are also mutated. If the wheel lands on a mutation the second time as well, the second selection is not considered. However, if it lands on crossover the second time, the individual may participate in crossover. If the crossover is selected (first or second spin), the individual is collected to the parent pool. After each individual has gone through the mutation/crossover selection and the pool contains all prospective parents, crossover is administered by simply collecting two individuals (parents) at a time from the pool and performing the crossover operation. The parents are not returned to the parent pool after the crossover, and thus each prospective parent can be used for one crossover only (per generation).

After the chromosomes are mutated (and individuals have mated), a corrective function is used to check that the architectures are legal. The corrective function checks certain "architectural laws". For example, when removing a pattern, the interface of an operation is set to 0. However, if the operation is required by an operation in another class, it should implement the interface provided by its class. Additionally, the function checks that all interfaces that are required by at least one class. The corrective function is also used to check that there are no contradicting patterns as a result of crossover. For example, it may be that operation o_k would be part of a Template Method pattern (i.e., o_k is placed in the "TemplateMethod"subclass of class I, class I contains o_i, and o_i requires $o_{k,}$) in the mother, but in the father o_k is placed behind a Strategy pattern. If the crossover point would now be between loci *i* and *k*, the patterns would not transfer to the offspring as whole, but the operations would be part of contradicting patterns. Thus, one of these patterns must be removed to make the architecture valid again.

3.1.4 Fitness and selection

At this point (mutations and crossover having been applied) the population is larger than in the beginning of the generation, as all the children produced by crossover have been added to the collection of chromosomes. The concept of natural selection now steps in, as the weakest individuals must be discarded from the population. This is done by first evaluating each individual and then performing selection.

As discussed in Subsection 2.1.3, evaluating software architecture quality is very difficult, and several viewpoints should always be considered. For

the GA the evaluation must be done using some kind of metrics, as the algorithm has to be able to straightforwardly order the architectures according to their quality. I have used three different quality attributes to evaluate the synthesized solutions: modifiability, efficiency (performance) and complexity. Modifiability is a natural choice, as the main purpose for which the design patterns are used is to increase the maintainability and modifiability of a system. Efficiency, in turn, is a counter attribute to modifiability, as many modifiability-enhancing modifications to architectures have a direct (negative) effect on efficiency. Thus, when examining the power of the algorithm, it is natural to concentrate on how much the algorithm manages to increase modifiability (from the zero state of null architecture). Having efficiency as a counter-weight makes sure that the algorithm is given some kind of restrictions, and can not wildly apply design patterns at every possible location. Complexity is used to ensure that modifiability has a contradicting quality attribute also in those cases where there is no clear efficiency drawback (e.g., in the case of applying a Template Method pattern).

Constructing the fitness function began by inspecting different software metrics and what properties they measure. The CK metrics (the ones suitable here, i.e., disregarding class hierarchies) were chosen as the base line for the fitness function. The CK metrics were chosen because applying them did not require any information of the semantics of operations, and clearly focused on class structure only. These metrics were then refined and extended so that they also consider the solutions that are not defined in the original metrics, such as the message dispatcher and server. While metrics were being defined, they were grouped according to whether they have a positive or negative impact on modifiability or efficiency. Complexity is evaluated with a simple formula. Thus, the actual fitness is composed of five sub-functions: positive modifiability, negative modifiability, positive efficiency, negative efficiency and complexity. Negative sub-functions are given coefficient -1, as they should always give values below zero. The total values for modifiability and efficiency are thus the sums of their respective sub-functions. Each sub-function is normalized so they have the same range. Each sub-function is also given a weight, so if one wants to emphasize one quality attribute over another, it can be assigned a larger weight.

The fitness function has constantly evolved during the research process. However, after several iterations, the fitness function is defined as follows. When w_i is the weight for the respective sub-function sf_i, the core fitness function $f_c(x)$ for solution x can be expressed as

$$f_c(x) = w_1 * sf_1 - w_2 * sf_2 + w_3 * sf_3 - w_4 * sf_4 - w_5 * sf_5.$$

Here, sf_1 measures positive modifiability, sf_2 negative modifiability, sf_3 positive efficiency, sf_4 negative efficiency and finally sf_5 measures complexity. The sub-functions are defined as follows (|X| denotes the cardinality of X):

sf_1 = (|calls to interfaces| * ∑ (variabilities of operations called through interfaces)) + (|calls through dispatcher|) ∗ ∑ (variabilities of operations called through dispatcher)) – |unused operations in interfaces| ∗ β ,

sf_2 = |calls between operations in different classes, that do not happen through a pattern|* ∑ (variabilities of called operations) + |calls between operations in same class|* ∑ (variabilities of called operations) * 2,

sf_3 = ∑ (|operations dependent of each other within same class| ∗ parameterSize) + ∑ (|usedOperations in same class| ∗ parameterSize + |dependingOperations in same class| ∗ parameterSize),

sf_4 = ∑ ClassInstabilities + (2 * |dispatcherCalls| + |serverCalls|) ∗ ∑(frequencies of operations called through dispatcher or server) + |calls between operations in different classes|,

sf_5 = |classes| + |interfaces|.

The multiplier β (β > 1) in sf_1 means that having unused operations in an interface is almost like breaking an architecture law, and should be more heavily penalized. It should also be noted that in sf_1, most patterns also provide an interface. In sf_3, "usedOperations in same class" means a set of operations in class C, which are all used by the same operation from class D. Similarly, "dependingOperations in same class" mean a set of operations in class K, which all use the same operations in class L. ClassInstabilities measures the relation of calls between and within classes [Amoui et al., 2006].

As stated, the fitness function has evolved during the research process. However, the core of the fitness function (division into five sub-functions) has stayed the same. Most alterations have been to coefficients and some minor calculations. One of the notable changes concerns the impact of the message dispatcher and its connections on positive modifiability (sf_1). In the experiments discussed in publications [I-V], [VII] and [VIII], sf_1 calculated the product of the (variabilities of) connections instead of the sum, as given above. This was changed in the final version of the fitness function, as in certain circumstances (discussed in Chapter 4), modifiability significantly overpowered the fitness value due to using the product of connections. Due to a typing error the fitness function formula

presented in publications [I-IV] suggests that the sum of (variabilities of) connections is used. In fact, the product of message dispatcher connections has been used in all other publications (with experiments) except publication [IX] and this thesis, where the latest version of the fitness function is used. This error, however, does not affect the presented results, which are based on complete (sub)fitness values, as the actual coded fitness function has been essentially the same for all publications [I-V], [VII] and [VIII].

It should be noted that all the patterns actually increase modifiability and decrease efficiency and complexity. The architecture is most efficient and simplest at null architecture stage, as there is minimal amount of classes and interfaces and connections between them. Also, no high-impact design choices, such as the message dispatcher, are present at this stage. Thus, when the synthesis progresses and patterns are added, efficiency and complexity values for the architectures decrease while the modifiability value increases (vice versa, if patterns are removed, modifiability will decrease while the other quality values increase).

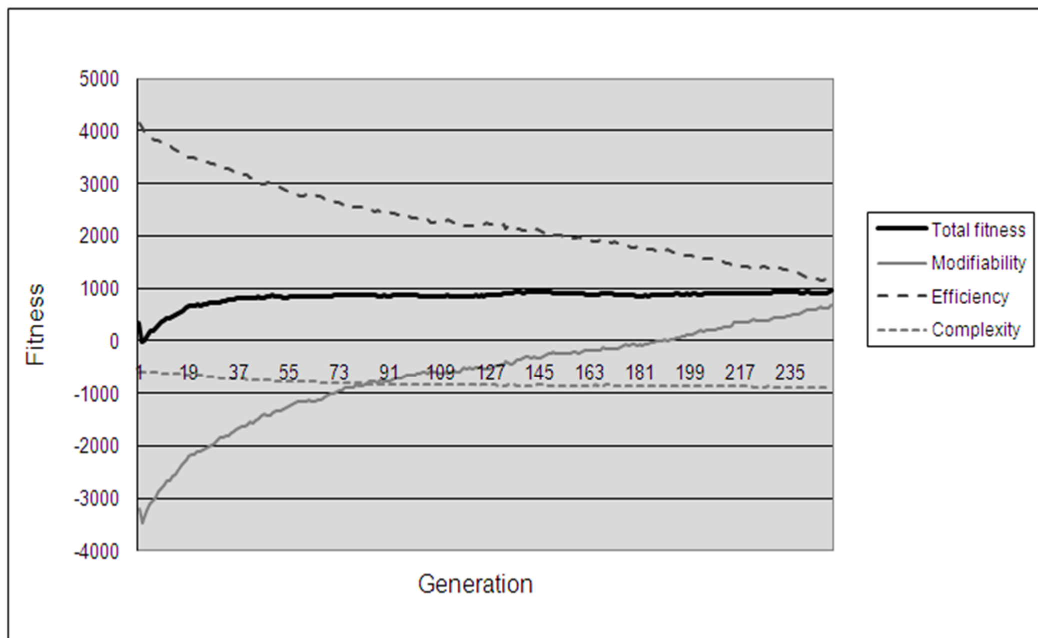


Figure 18. Example fitness graph for ehome

An example fitness curve for ehome achieved with the fitness function defined above is given in Figure 18. This is the average fitness curve of 20 runs, and displays how the average fitness of the ten best individuals (the elite) in each generation develops. The fitness curves for different quality attributes have been extracted to show what kind of balancing act it is to find architectures that have good quality from all viewpoints.

After each individual has been evaluated, they are ordered according to their fitness values. The very top of the population, the elite, is transferred

straight to the next generation. For the others a rank-based roulette wheel selection is made. Each individual is given a slice which corresponds in size to the rank the individual has in the population. A rank-based selection is more realistic, as the fitness function hardly gives absolute values in terms of how much "better" one individual is compared to another. After each spin of the wheel the selected individual is moved to the next generation and the sizes of slices are adjusted so that there are always as many slices in the wheel as there are individuals still left in the "old" population. After 100 (as many as in the initial population) individuals have been selected, the rest are discarded.

The process of mutation, crossover, fitness evaluation and selection is repeated for a defined number of generation; in this thesis 250 generations is used as the default length of an evolution. After the evolution is finished, the best solution of the last generation is given as a class diagram.

3.1.5 Case studies

I will briefly present results from the two case studies. The fitness graphs and example architectures presented here have been achieved with the fitness function and mutation specifications defined above. The case study results are intended to give an idea of what kind of fitness values the approach produces and how do the designs look like. Evaluation and validation of results is given in publications [I], [II] and [VIII] and in Chapter 5.

The fitness graph for ehome is given in Figure 19, and the fitness graph for robo is given in Figure 20. The graph for ehome is actually the same graph as in Figure 18; here the different sub-functions simply are not separated in order to make it easier to compare the ehome fitness graph to the one given for robo.

In both cases the fitness value plummets right after the evolution begins, and after this there is a rapid ascend. For ehome (Figure 19) the initial ascend in the curve is much bigger than for robo. However, after very fast development during the first 50 generations, the curve stabilizes, and although it does not seem to have come to a complete halt, there is not very much improvement.

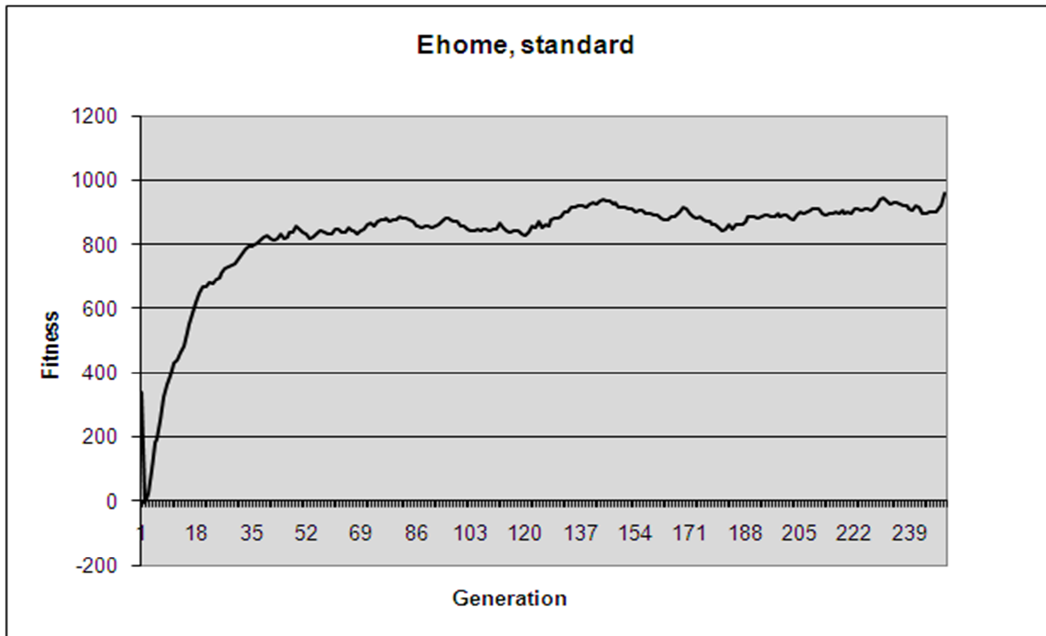


Figure 19. Fitness curve for ehome

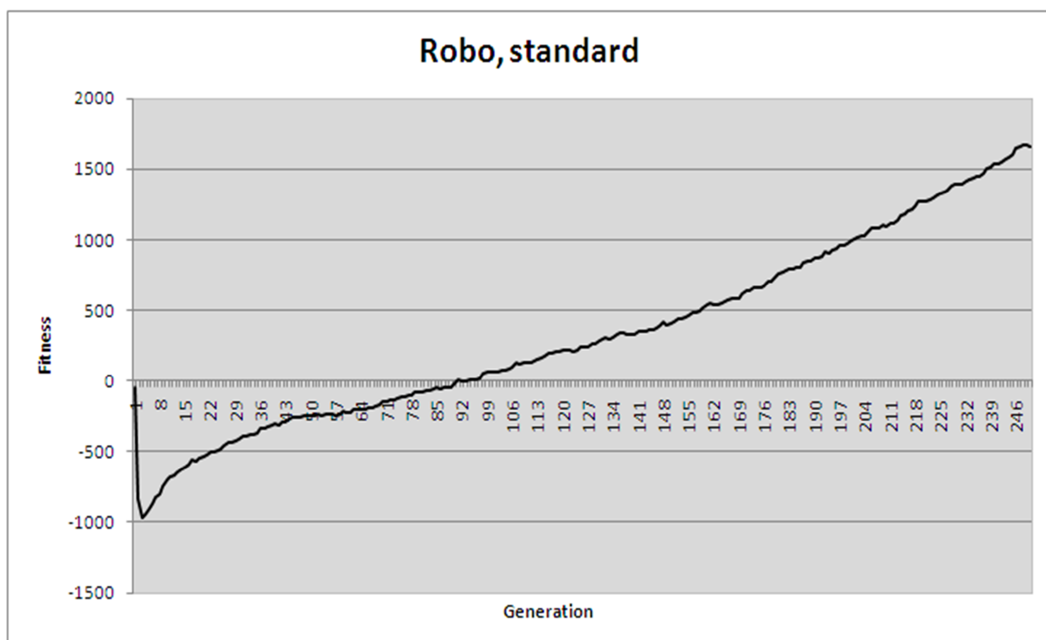


Figure 20. Fitness curve for robo

In Figure 20, however, the curve for robo develops quite differently. The initial ascend is much slower than for ehome, but the fitness keeps improving for the entire evolution, and actually seems to even speed up after 150 generations. However, in the very end the fitness curve seems to slightly stabilize.

An example architecture for ehome is given in Figure 21. This is not the actual class diagram produced by the synthesizer, as it would be too space-consuming and quite complex to interpret. The example is a simplified class diagram, where operations are omitted and the design

choices (patterns and styles) made by the algorithm are emphasized. The same kind of notation is used for all examples given from now on in this thesis. Intelligent selection of design choices is the purpose of the algorithm, and thus making the interpretation of the example diagram easy will best serve the purpose of examining how the algorithm works.

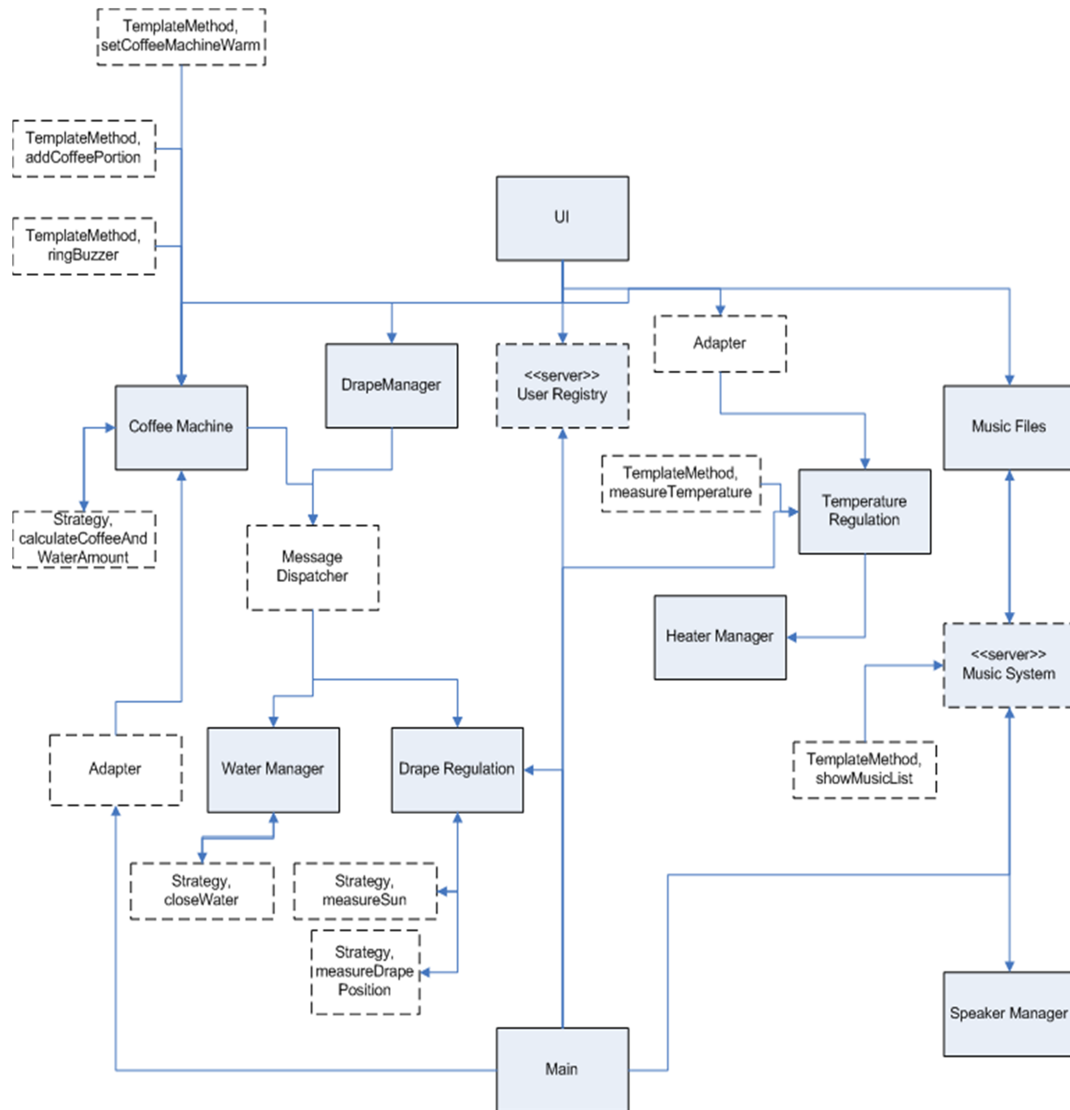


Figure 21. Example architecture for ehome

The example in Figure 21 is quite a typical architecture achieved with the standard method, where all quality attributes (sub-fitnesses) are weighted equally. As can be seen, the message dispatcher architecture style is present in the architecture, but it is used very little; only the CoffeeMachine/WaterManager and the DrapeManager/DrapeRegulation connections are handled through the message dispatcher. Also, the client-server style is present; in reality, the simultaneous existence of these two architecture styles would probably not be accepted. In the example there are instances of all the lower level design patterns. The large number of TemplateMethod pattern instances is quite common for ehome solutions.

All solutions also usually contain some Adapter patterns, while the amount of Strategy patterns varies; even though this particular example has several Strategies, some solutions achieved with the same parameters do not have any Strategy instances.

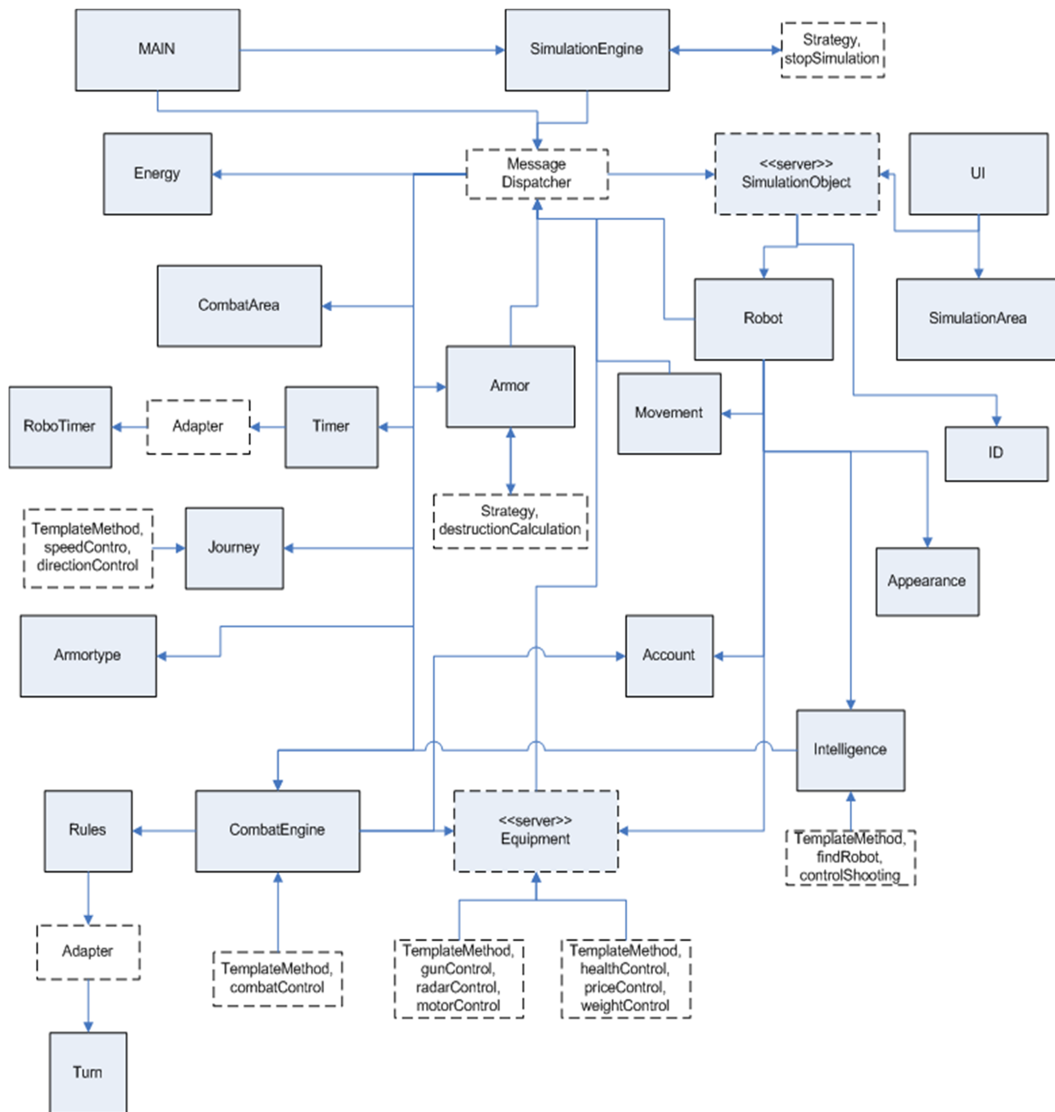


Figure 22. Example architecture for robo

An example architecture for robo is portrayed in Figure 22. Here the message dispatcher is also present, and although its full capacity is still not used, the level of communication through the message dispatcher is significantly higher than in the case of ehome. This could be partially due to the restriction that only operations in different null architecture classes may use the message dispatcher for communication (robo has twice the number of classes already at null architecture level), but this does not completely explain why the GA has so much favored the message dispatcher. Again, like with ehome, the client-server style is introduced alongside the message dispatcher architecture style. In the case of robo, there are also several instances of the Template Method pattern, and few

instances of Adapter and Strategy. This type of structure is quite common for robo.

To summarize, when using the standard parameters and no quality attribute is favored over another, the test results for both cases are quite similar. The message dispatcher is present in the system, but only handles a part of the communication between different classes. The usage of message dispatcher appears stronger in the case of robo. For both cases, the client-server architecture style is used in addition to the message dispatcher style; a human designer would probably use only one or the other. As for the design patterns, solutions for both cases usually contain several instances of Template Method patterns, and some instances of Adapter and Strategy patterns. No instances of the Façade or Mediator pattern were found for either case.

3.2 TOOL SUPPORT

As the presented research is mostly based on conducting a large number of experiments, a quick and easy to use practical tool is required. I will here describe the tool only briefly, as it is only meant to assist in applying the synthesis, and the main contribution of this thesis is providing the methodology for the synthesis itself. The tool essentially provides a user interface for defining the requirements, easily adjusting different parameters for the algorithm and viewing the results – both the fitness graph and the end result in the form of a class diagram.

The tool "Darwin" is an Eclipse [2011] plug-in and uses UML2Tools [2011] to produce the class diagram in the end as well as in the requirement specification phase. A plug-in called JFreeChart [2011] is used to display the fitness graphs. The tool is based on a "GA engine", which essentially is the algorithm (synthesizer) described in the previous section. A user interface is provided for specifying the requirements, after which they are given to the algorithm.

Requirements can be given as encoded files, similarly as in the original implementation. However, the functional requirements of the system under design can also be defined with the tool itself. The tool provides a user interface for creating use cases and sequence diagrams, and provides automated support for transforming the sequence diagrams into the class diagram representing the null architecture. After the null architecture is somehow specified (either with an input file or through specifying use cases with the tool), a simple press of a button is required to start the evolution for synthesis.

The algorithm provides the end result (class diagram) which is converted into an editable form and displayed by the tool. The author's contribution

to the tool is the GA engine (and interfacing it with the tool) as well as participation in designing the tool. Other members of the project team are responsible for implementing the actual user interface and interfacing with Eclipse, UML2Tools and JFreeChart.

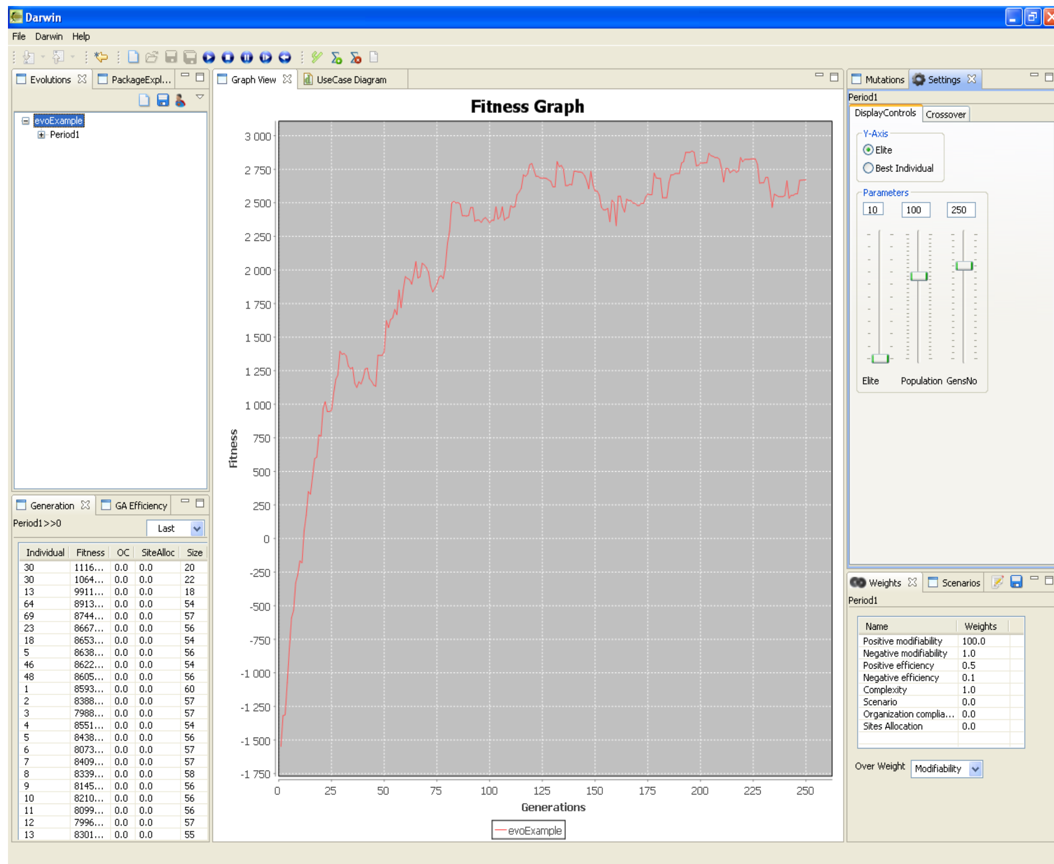


Figure 23. Screenshot of Darwin tool portraying a fitness graph

A screenshot of the tool is given in Figure 23, which shows the results of an example run with ehome. In the lower left corner is the information regarding each individual in a population. One can select the desired generation, and the fitness values for the respective population are displayed. Any individual can be selected for viewing; this way the transformations made to the architecture can be backtracked.

In the lower right corner one can set the desired weights for all subfunctions. In this example, modifiability has been given an exceptionally large weight, which is clearly portrayed in the fitness graph, as it makes quite fast improvement in the beginning. Note, that in this example the fitness graph is the result of only one run. The tool also provides options to view all the different sub-fitness graphs at once and to create an average fitness graph from several runs. In the upper right corner there are two tabs: Mutations and Settings. In the Settings tab (open in the figure) one can set the number of elites, the size of population and

number of generations. In the Mutations tab the probabilities for different mutations are set.

A screenshot with a different view of the tool is given in Figure 24. In this case the best individual of the final generation is selected for viewing. Here the UI and Control classes call the Dispatcher, which in turn calls the UserRegistry and the TemperatureControl interfaces. An instance of the Strategy pattern can also be seen, as one of the operations in UserRegistry has been separated and placed behind a Strategy interface. The class diagram is completely editable, contrary to the class diagram given by the original implementation, which was in picture format.

The tool essentially eases using the synthesizer, as all information can be given through the user interface. Much more information on the development of solutions can also be provided to the user, and a live viewing of the architecture development can be made possible. The tool and its implementation are discussed in more detail in publication [V].

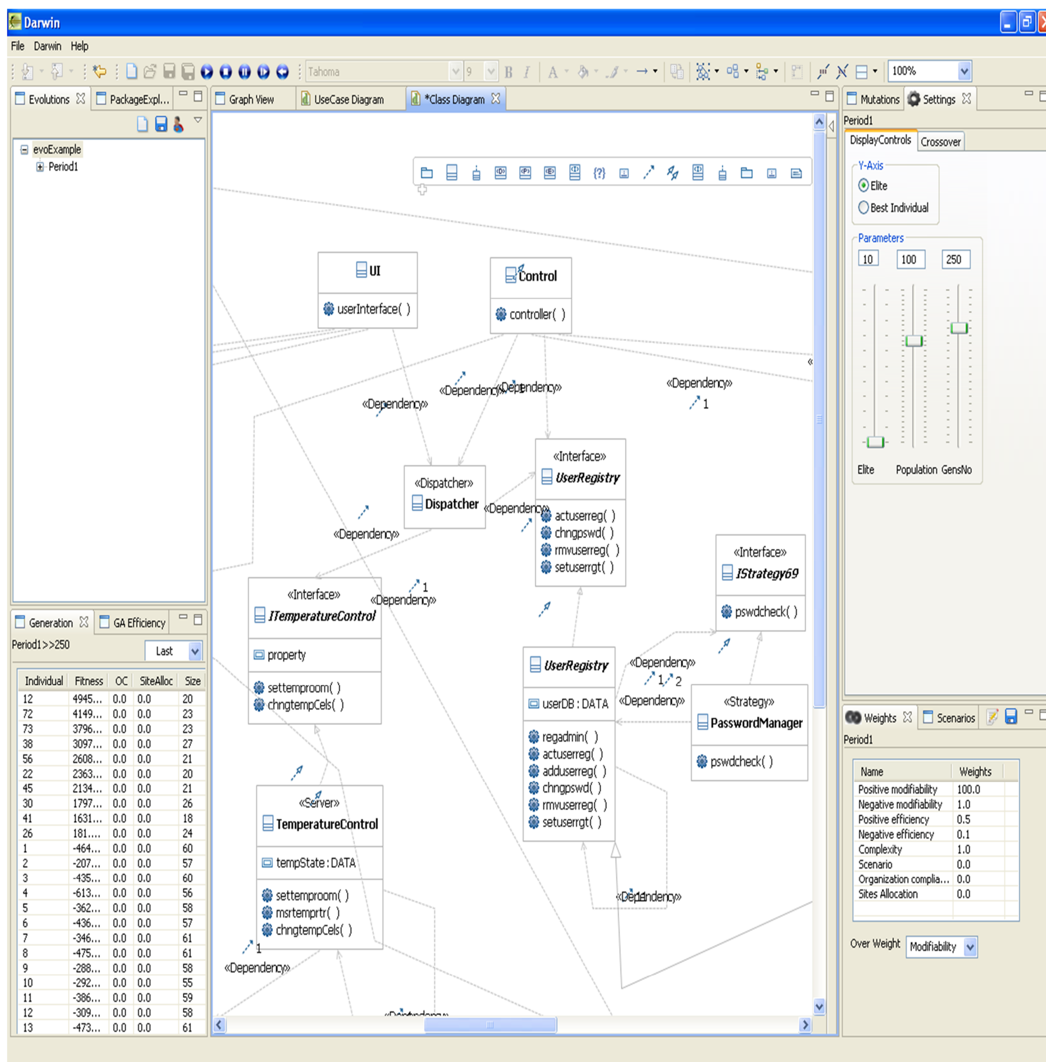


Figure 24. Screenshot of Darwin tool showing an example solution

4 Variations and Extensions

As discussed in Chapter 1, it is not given that a standard, “simple”, implementation of the GA would be best for such a complex problem as software architecture design. Thus, several variations to the basic method described in the previous chapter were made and experimented with.

The crossover operator is essential in GAs, as it differentiates GAs from local searches by enabling large jumps within the search space from one solution to another. Thus, correctly defining the crossover operator is critical in order to thoroughly explore the search space and avoid getting stuck in local optima. A simple single-point random crossover did not seem sufficient for the problem of software architecture synthesis, and thus two variations for the traditional crossover were examined.

Firstly, one option for crossover was to discard it completely, and use some other method for reproduction. In real world, no two architects would randomly swap parts of their designs, but rather develop alternative designs independently. Thus, in publication [IV] it was experimented whether crossover was required at all. This would more accurately follow the real-life procedure where one architecture design is developed at a time, and if there are competing designs, only one is selected, and it is selected as a whole.

Secondly, a logical alternative to completely discarding the crossover would be to make the crossover operator more intelligent, and swap parts of architectures more purposefully. This complementary crossover can operate on two levels. One way is to only select designs which as a whole fulfil competing quality requirements, and then randomly combine them in the effort to produce an all-around good design. Another way is to inspect the architectures in more detail, and purposefully combine distinct

(best) parts of the competing designs. These two versions of complementary crossover were implemented and experimented with in publication [VII].

Just as crossover is critical in exploring the search space, the fitness function is critical in order for the algorithm to find the correct solutions. If the fitness function is poor, the algorithm will search for the "wrong" solutions, and the outcome will be just as wrong. Thus, while the single weighted fitness function presented in Subsection 3.1.4 is shown to measure all the right things, the problem of architecture design just seems too complex to evaluate with simply a combination of metrics.

Firstly, as discussed in Section 2.1, evaluating architectures is extremely difficult, and thus it is no surprise that a single weighted fitness function is very hard to calibrate so that it would produce satisfactory results. Thus, in publication [III] it was studied whether adding more specific information of possible modifiability needs would help in making more detailed decisions.

Secondly, as a single weighted fitness function provides minimal answers to how a solution fulfils specific quality requirements, and it would be good to have several candidate solutions instead of just one, a multi-objective approach was implemented with Pareto optimality. Pareto optimality enables producing several potential solutions and evaluating each quality attribute separately. This multi-objective approach and initial results achieved with it are discussed in publication [IX].

In this chapter I will discuss the variations and extensions described above. Results from case studies are discussed briefly; further evaluation is given in the publications and in Chapter 5. Note that the setup for experiments made with different variations differ slightly from that presented in the previous chapter (and from each other); small adjustments to the fitness function, the mutation probabilities and some restrictions to mutations have been made during the research process.

4.1 ASEXUAL REPRODUCTION

4.1.1 Method

As stated, in a real-life design situation it would be quite abnormal for two software architects to come up with completely different suggestions for a given system and then start randomly swapping parts of the designs. Only one architecture is usually designed, and it is iteratively developed until it meets given quality requirements sufficiently well. Thus, it seemed logical to experiment whether simply applying mutations would be sufficient in order to achieve synthesized architectures with reasonable quality.

While the main idea was not to use crossover, the concept of natural selection which is essential in GAs should still be kept. In order to perform natural selection, the size of the population should be grown so that there would actually be “extra” individuals that could be discarded. As crossover would not be present to produce offspring, increasing the population size should be done in some other way.

The problem was solved using *asexual* reproduction. In nature there exist cases where individuals of certain species can change sex and thus reproduce by themselves. The same ideology was used here. The implementation for a GA relying solely on mutations and asexual reproduction, and not using crossover, was made so that in the beginning of each generation every individual was cloned twice (resulting in three clones of the same individual). This triple-sized population was then subjected to mutation. Natural selection now had very good grounds, as the population was much larger than ever accomplished with crossover. The same elitist selection combined with rank-based roulette wheel selection was used as described in Subsection 3.1.4. Otherwise, the implementation was similar to the basic method, i.e., it used the same mutations and same algorithmic procedure.

4.1.2 Case studies

Results from the two case studies showed that asexual reproduction performed much faster than the regular crossover-based method, as can be expected. The fitness curves ascended much sooner and the ascent was also very steep. However, the curve seemed to settle to its optimum very soon: no improvement could be seen after the first 100 generations or so. Even tests with increased number of generations looked the same; after 100 generations the fitness curve simply remained flat. The fitness curves for ehome and robo are given in Figures 25 and 26, respectively, and were calculated as averages of 10 runs.

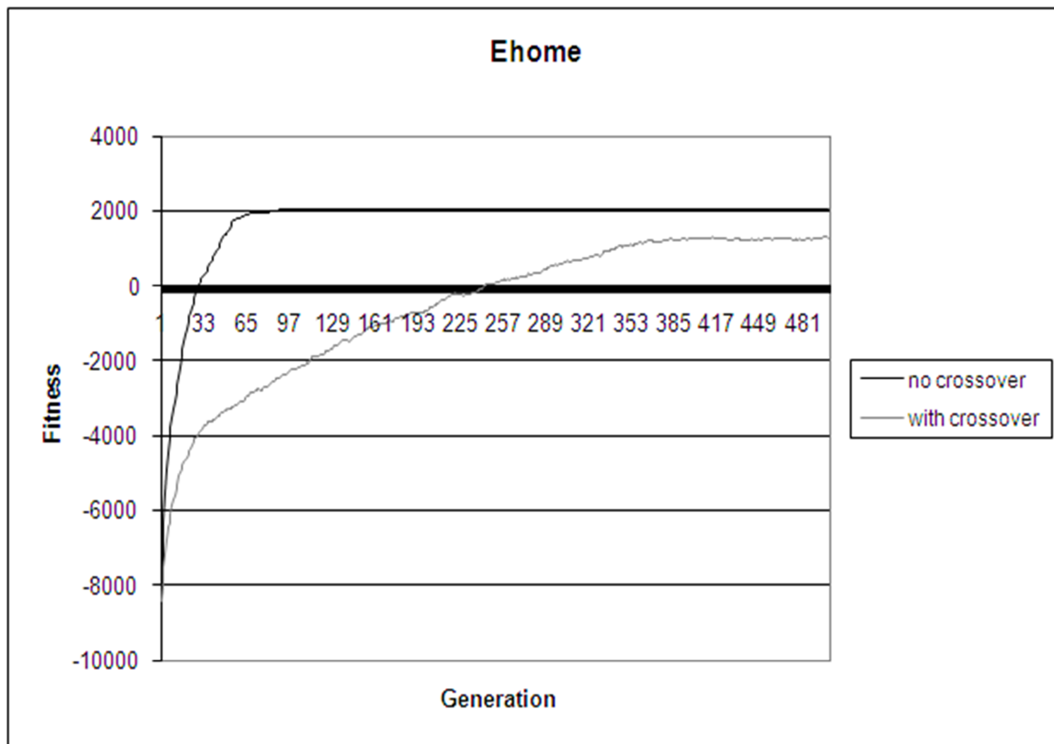


Figure 25. Fitness curve for ehome, asexual reproduction

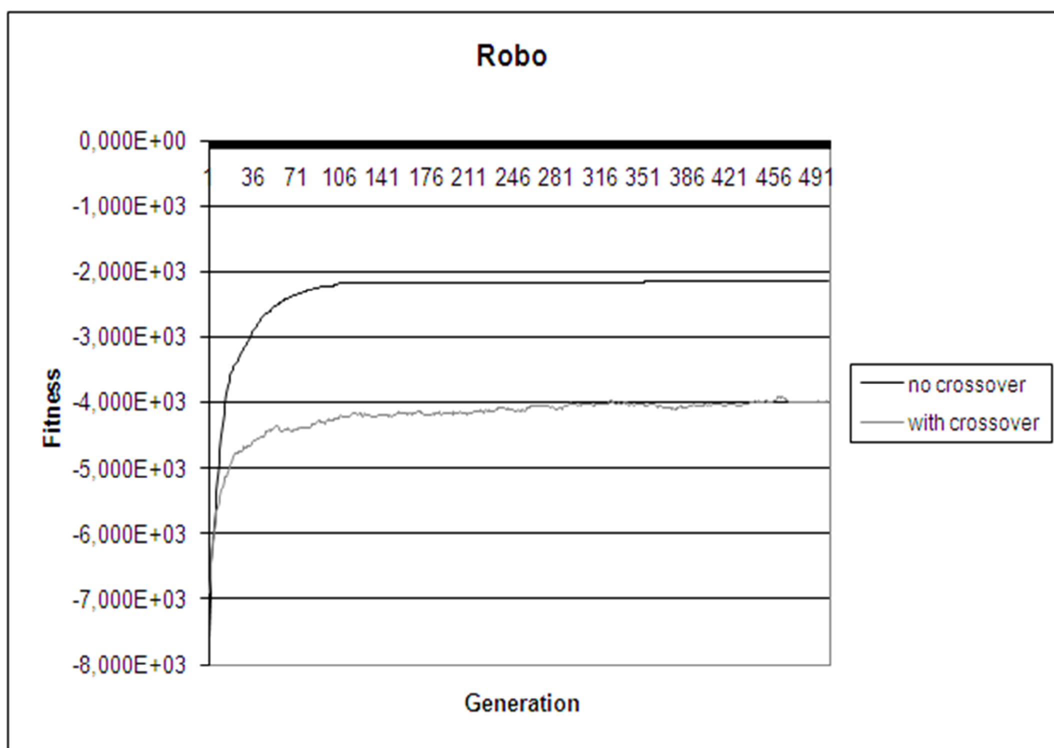


Figure 26. Fitness curve for robo, asexual reproduction

Thus, it could be concluded that while asexual reproduction was faster in the beginning, it failed in thoroughly exploring the search space and quickly landed to a local optimum. While further examining the fitness data to find reasons for this, a curious phenomenon was discovered: in the

last populations there was very little or no variance in the fitness values of the top half of the population. Thus, it would appear that in the end the population mostly consists of clones of the same solution. This is a result of the elitist selection keeping the best individuals (eases keeping several clones of the best solution), having also included the "null" mutation and having quite strict preconditions for mutations. If the preconditions for a mutation are not met, the mutation cannot be applied, and in effect, several solutions stay the same for many generations. When they are cloned several times, the end result is, indeed, a population of clones.

When the actual solutions were studied, it was clear that the GA had favored a very limited amount of design choices. The solutions did not contain any instances of the message dispatcher style (the client-server style was scarcely used as well) and relied heavily on the Template Method design pattern. Another design pattern that was commonly used was the Adapter, while Strategy was hardly ever found. The Adapter pattern is easier to apply than Strategy, as the precondition is looser. The Template Method, in turn, penalizes efficiency and complexity the least. Thus, also the solutions indicate that asexual reproduction is unable to effectively investigate the search space, and goes about building the architectures quite single-mindedly. Typical examples of the resulting architectures for ehome and robo are given in Figures 27 and 28, respectively.

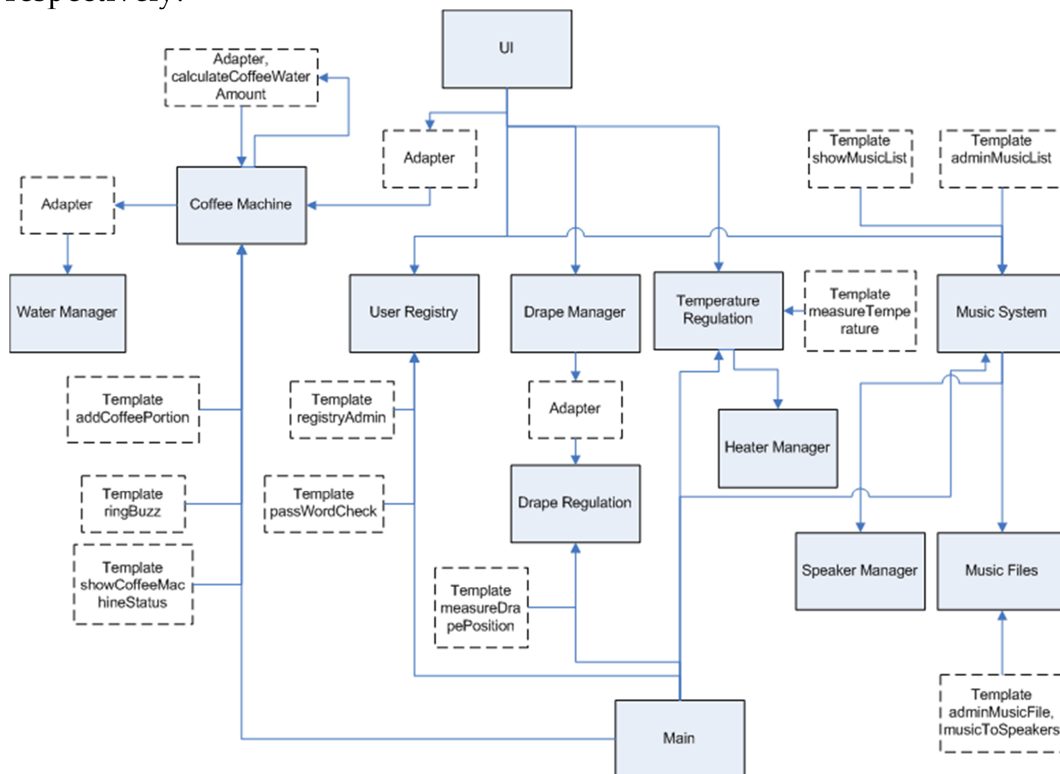


Figure 27. Example solution for ehome, asexual reproduction

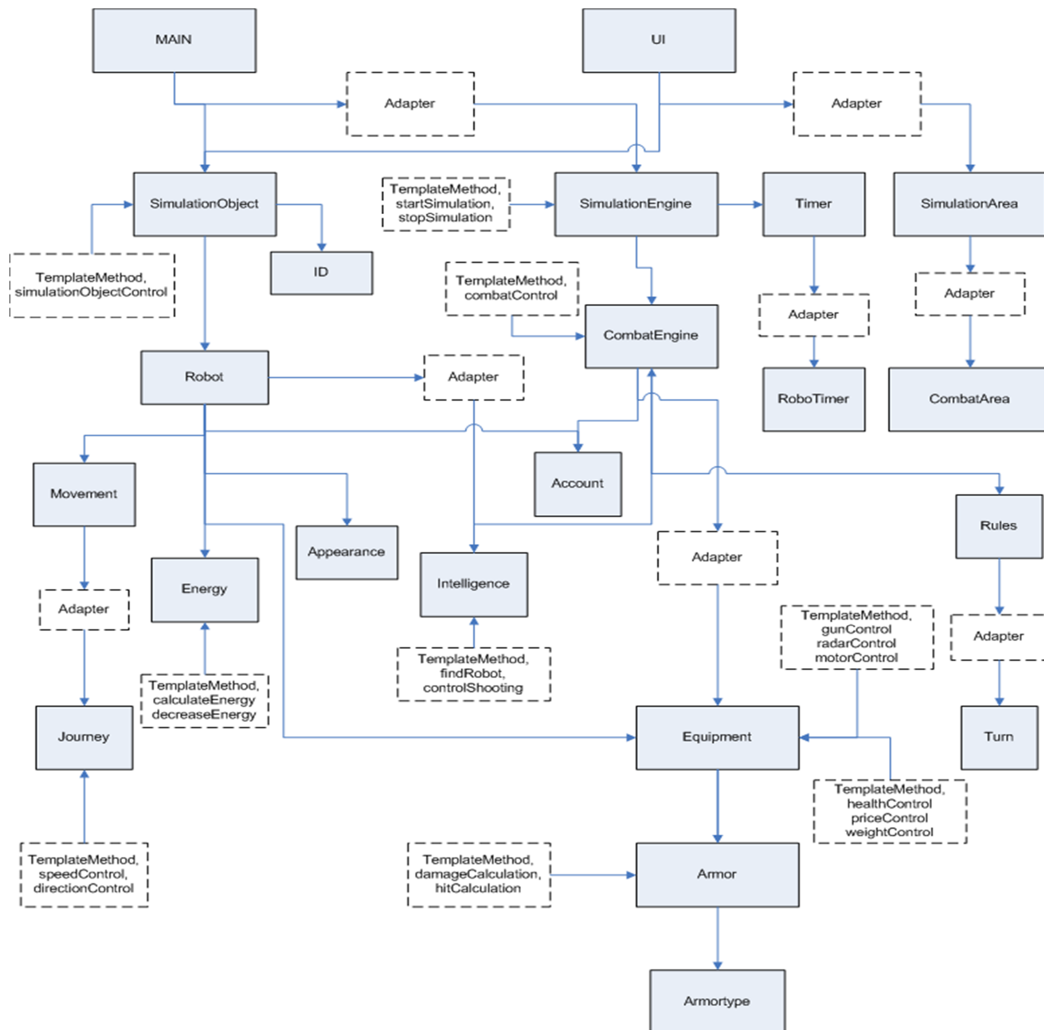


Figure 28. Example solution for robo, asexual reproduction

To summarize, asexual reproduction does not use crossover, but only transforms the architecture with mutations. In order to enable natural selection, individuals are cloned to create a sufficiently large population, after which the strongest are chosen with roulette-wheel selection. Results indicate that this approach leads to faster evolution but quickly lands on a local optimum. This variation to crossover is discussed in more detail in publication [IV].

4.2 Complementary crossover

4.2.1 Method

As asexual reproduction did not produce the desired outcome, and the random crossover is not very close to a real-life design situation, the idea of enhancing the crossover operator so that it would act in a more purposeful way seemed appealing. Thus, the *complementary crossover* and the *complementary gene-selective* crossover were implemented. Biologically, complementary crossover corresponds to the theory of genetic

compatibility [Zeh and Zeh, 1996], i.e., that some individuals are genetically more compatible with each other than others. Genetically compatible individuals are more likely to produce viable offspring, and partners are sought so that the individuals have complementing properties. Thus, their offspring would inherit as many desirable properties as possible. In software design, the complementary crossover corresponds to a situation where two architects exchange ideas on their designs, and attempt to combine the best parts of two competing solutions.

In simple complementary crossover the objective is to purposefully combine two parents which satisfy different quality requirements. Modifiability and efficiency were chosen as the pair of competing quality attributes. In the beginning of each generation, the population was ranked based on both modifiability and efficiency. If an individual's modifiability rank was higher than its efficiency rank, it was considered that the individual satisfied modifiability related requirements better (and vice versa in terms of efficiency). The modifiable individuals were labeled as mothers and the efficient individuals were labeled as fathers.

After the population was divided into mothers and fathers, the mutation operation and parent-selection for crossover were performed normally. However, when crossover was actually applied, the individuals in the parent pool were inspected more thoroughly. The fathers were placed in a subpool of their own, and the mothers similarly to their own subpool. The crossover was then administered by choosing a pair of parents so that one came from the father pool and one from the mother pool. If one pool had more individuals than the other, these remaining individuals were not used for crossover. The actual crossover point, however, was still chosen randomly. The ideology was that these two individuals with different strengths would complement each other and produce offspring which fulfills both quality attributes.

As the crossover point in this simple complementary crossover was chosen randomly, there was the risk that the best parts, i.e., the parts of individuals that have the biggest impact in satisfying the quality requirements, might be lost. Thus, a more intricate version, the gene-selective complementary crossover, was implemented.

When using the gene-selective complementary crossover, the ranking and pooling of parents was done similarly to simple complementary crossover. However, when the actual crossover was performed, the crossover point was no longer selected randomly. In order to find the best crossover point, the most modifiable section of the mother and the most efficient section of the father were sought. These sections were identified by using the maximum contiguous sub-sequence sum [Weiss, 1998]. Each supergene was calculated a value by how much it influenced the modifiability or

efficiency value. In other words, the operation defined by that particular supergene was inspected in terms of its impact on the different sub-fitnesses. The best sequence of supergenes was selected as the modifiable or efficient section.

The crossover point was then selected randomly from the indexes that were in between the found sections. Only one individual was produced, and that individual was thus combined so that it contained the best sections of both its mother and its father. The gene-selective complementary crossover is depicted in Figure 29. Here index l is the starting index of the most modifiable section of the mother, while k is the end index. For the father, m is the start index of the most efficient section, and p is the end index. The crossover point cp should now land between k and m in order to retain both sections for the child.

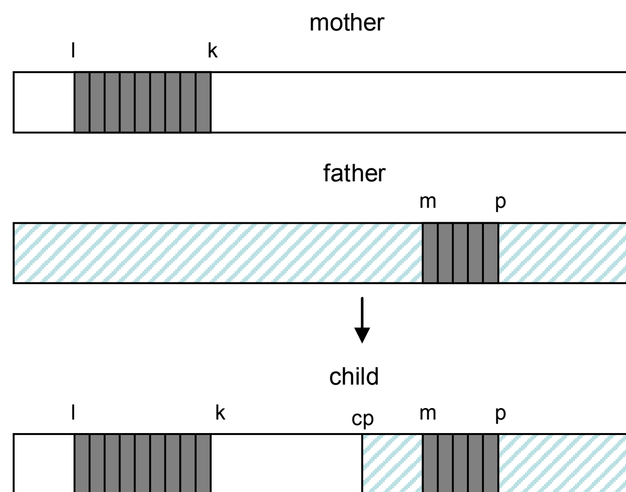


Figure 29. Gene-selective complementary crossover

4.2.2 Case studies

The fitness curves for both cases are portrayed in logarithmic scale. The fitness function used in this study (as well as for asexual reproduction) was able to give exponential reward for the use of message dispatcher, if it was used very heavily. The message dispatcher was rewarded in this way as it is not fully beneficial until it is used throughout the whole architecture, and not only between a few operations. Once the usage is intensive, it makes the architecture much more modifiable on a general level than individual design patterns. As the complementary crossover made such usage possible, the fitness values also developed exponentially, and logarithmic curves were required to evaluate the development of fitness values. Fitness values were calculated as averages of 20 runs.

Figure 30 illustrates the fitness curves for ehome. The curve for the standard method remains quite unchanged at just above 1000. Some development did happen, but not so much that it would show on logarithmic scale. The curves for complementary crossover actually descend for the first 100 and so generations, after which they start ascending, and reach quite high values.

Figure 31 shows the respective fitness curves for robo. Here the standard curve behaves similarly as in the case of ehome, while the difference between standard and complementary crossover curves is drastic. Because the difference between the two cases was so big, tests with 750 generations were made. These tests (discussed further in publication [VII]) showed, that with this longer evolution, ehome was also able to reach values as high as robo already did at 250, and the shape of the curve resembled much more those of robo as well. The fitness curves for longer evolution with ehome are given in Figure 32. The fitness curves for robo with longer evolution (750 generations) did not differ significantly from those achieved with shorter evolution (250 generations).

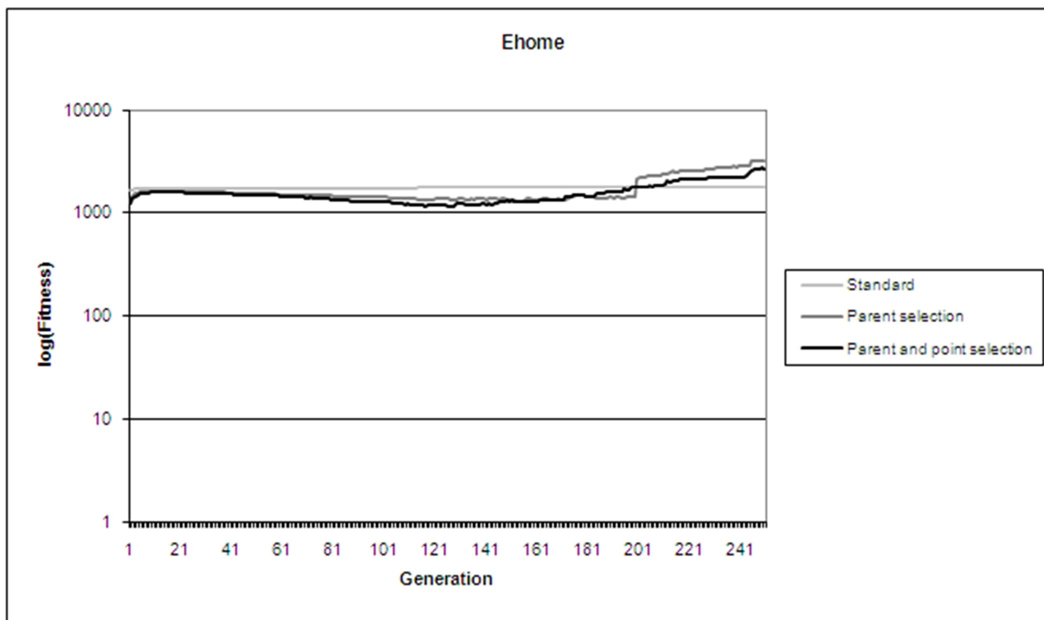


Figure 30. Fitness curves for ehome, complementary crossover

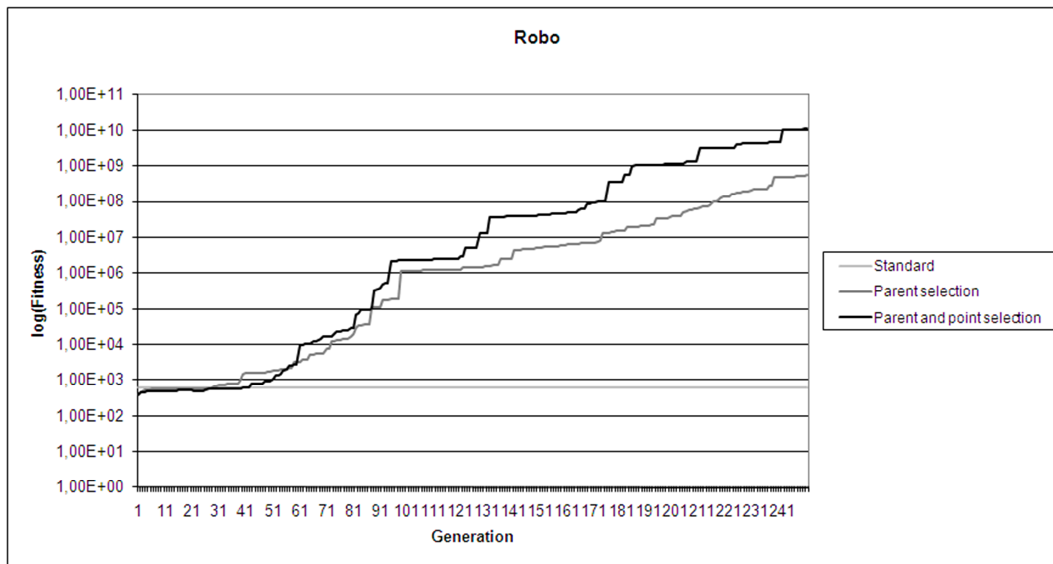


Figure 31. Fitness curves for robo, complementary crossover

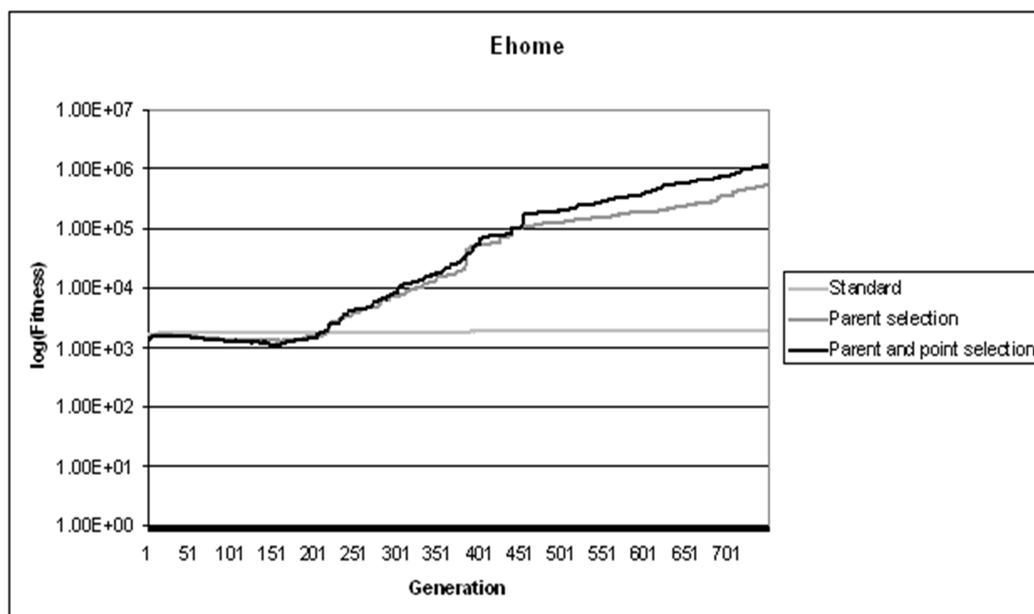


Figure 32. Fitness curves for ehome, complementary crossover, 750 generations

The architecture proposals achieved with complementary crossover illustrated quite well why the fitness curves behaved the way they did. The main difference between solutions achieved with complementary crossovers and the standard crossover was the presence and level of usage of the message dispatcher. When the standard random crossover was used, no solution contained the message dispatcher for either case. However, when complementary crossover was used, 18 of the 20 solutions for simple complementary crossover in ehome already contained the dispatcher, and in solutions with gene-selective crossover for ehome, and both complementary crossovers for robo, the message dispatcher was present in all the solutions.

The example solutions for ehome and robo, given in Figures 33 and 34, respectively, portray typical solutions with gene-selective complementary crossover. The example solution for ehome has been achieved after 750 generations, while the example for robo is the result of a 250 generations long evolution. It seems ehome is slightly more difficult to deal with from the algorithm point of view, and it takes a longer time to effectively use the message dispatcher. This explains the descend of fitness curves in the beginning: when the message dispatcher is present in the architecture but very poorly used, the penalty for it is much larger than the reward, and thus the fitness curve descends.

Both solutions have very centralized use of the message dispatcher, and also several instances of the different design patterns. The longer evolution for ehome seems to have affected also the number of patterns, as all low-level design patterns have been brought to the system, while for robo there are no Adapters, and the amounts of Template Methods and Strategies are significantly lower than for ehome.

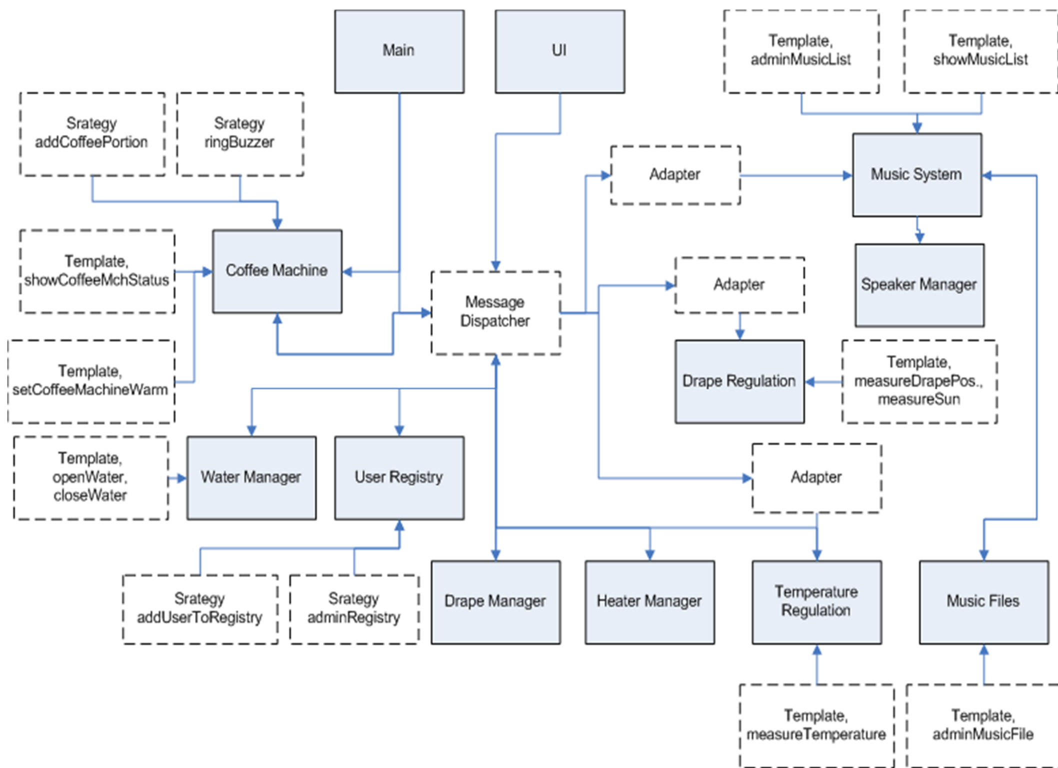


Figure 33. Example solution for ehome, complementary crossover

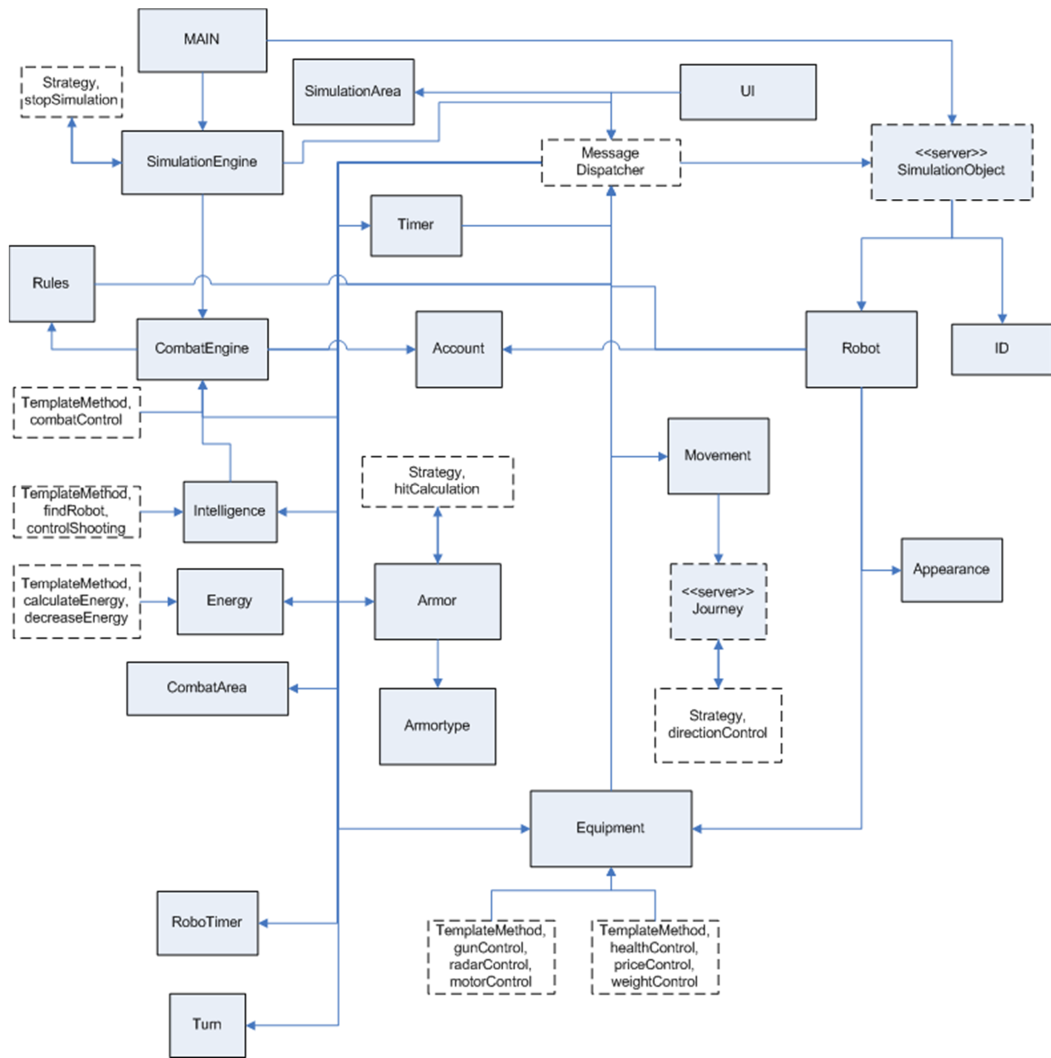


Figure 34. Example solution for robo, complementary crossover

To summarize, the simple complementary crossover combines parents which fulfill different quality requirements. The gene-selective complementary crossover similarly seeks complementary parents, but also inspects the parents and only produces one offspring with the best parts of the two parents. The complementary crossover is able to produce solutions with delayed reward, and the difference to simple random crossover is significant (to the better). This variation is discussed in more detail in publication [VII].

4.3 Scenarios

As discussed in Subsection 2.1.3, scenario-based evaluation, ATAM in particular, is one of the most popular methods for evaluating software architecture quality in practice. Thus, to ease evaluating modifiability (which is more difficult to elicit from the class diagram by metrics), the core fitness function was extended with a scenario-based sub-fitness function. This enables giving modifiability related scenarios where the

changes are targeted to certain operations. Evaluating the correct placement of patterns is thus more specific and the results more accurate.

4.3.1 Method

In ATAM or any other scenario-based evaluation methodology the scenarios are usually specified in free-form. Thus, the first thing to do in order to enable the GA to use scenarios is to develop an encoding for different types of scenarios. In this work, I have concentrated on modifiability-related scenarios, and have divided them into two main categories: *changing* scenarios and *adding* scenarios. A change scenario indicates that a function will be changed in some manner, and an adding scenario indicates that alternative implementations for a functionality will probably be given.

Each scenario is also labeled as *dynamic* or *static*. A dynamic scenario should be executable during runtime, i.e., it should not require changes to the code, and portrays changes from the user viewpoint. A static scenario should be executable during implementation, i.e., it may require small changes to the code, and concerns changes from the developer's point of view. Additionally, for each change scenario it is defined whether the change concerns *implementation* or *semantics*. Scenarios concerning implementation are more common, and they simply predict that the implementation of an operation is changed. Scenarios concerning semantics are rarer, and indicate that an operation may be changed to something completely different. Additionally, each scenario is given an estimated probability, which roughly describes how critical it is.

Each scenario is defined for one operation only. Thus, when encoding the scenarios they contain information about which operation they concern, about the three different classifications discussed above, and their the estimated probability. For example, a scenario for ehome could be "the developer should be able to change the way the optimal positioning of drapes is calculated with a probability of 80%". This scenario concerns development time changes, and is thus labeled as *static*, and requires the change of implementation, and is thus labeled as a *changing* scenario, which in this case concerns *implementation*. The encoded format of this scenario thus contains the name of operation concerned (calculateDrapePosition), the defined labels, and the probability (80%). The collection of encoded scenarios is given for the *scenario interpreter* as a text file. The scenario interpreter transforms the scenarios into a further encoded format for the synthesizer to process.

From the GA point of view, a mechanism is needed to evaluate how well the architecture responds to the scenarios. Evaluation is made by ranking

the possible design solutions (i.e., design patterns and general structural solutions) according to their suitability for each possible scenario type (based on all three categorizations), i.e., each scenario type is given a preference list of suitable design decisions. For example, in the case of a statically changing implementation scenario, the best way to prepare for this kind of modification would be to use a Template Method (if applicable), which would now be placed on top of the preference list. The second best way would be to use a Strategy pattern, in which case the new implementation could be offered as alternative algorithm. Using Strategy would now be second on the preference list. Other design solutions would similarly be examined and ordered in the preference list accordingly. It should be noted that no guidance is given for particular scenarios, as the rankings are only defined as per scenario categorization – not according to individual scenarios. The preference list is handled by the scenario interpreter.

When the GA calculates a scenario fitness value, it goes through the given list of encoded scenarios and compares the current solutions in the design at hand to the preference list provided by the scenario interpreter. The design is then rewarded points from each scenario; the points are scaled so that the higher the sub-solution regarding the scenario is in the preference list, the more points are rewarded. The actual scenario fitness value for one solution is achieved by simply summing the points gained from each scenario.

Formally, the scenario sub-fitness function sf_s can be expressed as

$$sf_s = \sum scenarioProbability*100/scenarioPreference.$$

Adding the scenario sub-fitness function to the core fitness function would result in the overall fitness, $f(x) = f_c(x) + w_s*sf_s$. As the scenario sub-fitness function measures modifiability, it may also be used to replace the modifiability functions sf_1 and sf_2.

4.3.2 Case studies

For both cases (ehome and robo) 15 different modifiability scenarios were invented based on the assumed evolution of these kinds of systems.
For example, scenarios for ehome include:

- the user should be able to change the way the music list is showed (scenario probability 90%)
- the developer should be able to change the way water is connected to the coffee machine (50%)
- the developer should be able to add another way of showing the coffee machine status (60%).

Scenarios for robo include:

- the player should be able to change the appearance of a robot (80%)
- the developer should be able to add an alternative armor control (80%)
- the developer should be able to change the implementation of the robot intelligence control (90%).

These scenarios are then analyzed to fit the classification framework of scenarios discussed above. The first ehome scenario is a changing scenario that regards implementation and should be handled dynamically. The second ehome scenario is a changing scenario that should be handled statically and concerns the semantics of the component, i.e., the interface may need to be changed. The final ehome scenario is a static adding scenario concerning implementation.

The first robo scenario is a changing scenario that should be handled dynamically and concerns implementation. The second robo scenario is an adding scenario that should be handled statically and again handles implementation. The last robo scenario is a changing scenario to be implemented statically.

When inspecting the fitness curves, the effect of adding the scenario sub-fitness function was evaluated in two different ways in the case studies. Firstly, the standard fitness curve with only the core fitness function was evaluated against the fitness function where the scenario fitness was added. The fitness curves for ehome and robo illustrating the difference that adding the scenario fitness makes are given in Figures 35 and 36, respectively. As can be seen, the curve where the scenario fitness is added naturally achieves higher values, as scenario sub-fitness may only increase the overall fitness value. However, the graphs also show that the time period where most of the development is made in the beginning of the evolution is 10 to 20 generations longer when the scenario fitness is applied. It appears that adding scenarios and thus enabling more detailed evaluation prevents the algorithm from converging to a local optimum too soon.

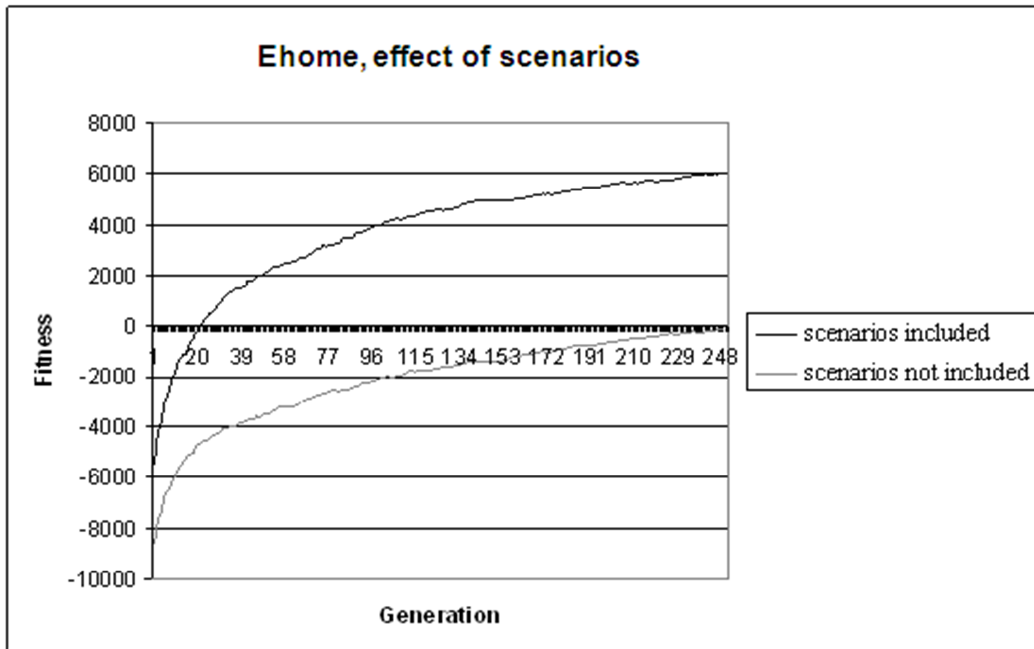


Figure 35. Fitness curves for ehome, total fitnesses with and without scenarios

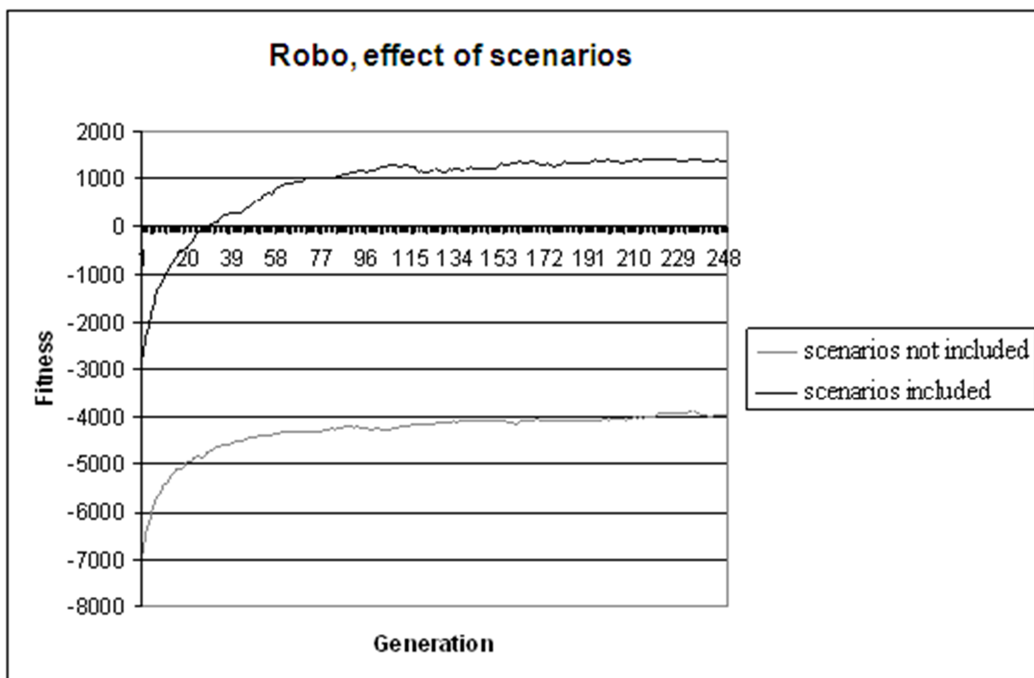


Figure 36. Fitness curves for robo, total fitnesses with and without scenarios

Secondly, it was tested whether the GA would be able to introduce scenario satisfying solutions if it was unaware of the scenarios. This was experimented by calculating the scenario fitness value but not using it for the GA evaluations (curve "scenario not calculated"), and comparing these fitness curves to the ones where the scenario fitness value was given for the GA to be used in evaluation. These curves for ehome and robo are given in Figures 37 and 38, respectively. These curves illustrate the

scenario fitness only, and does not take into account the other sub-fitness functions.

As can be seen, the results are quite similar to those already seen in the case where complete fitness curves where compared. When the scenarios are used for evaluation, the fitness curve has a longer period of rapid development, and achieves overall higher values than when scenarios have not been taken into account. Also, in the case of ehome (Figure 37), the scenario fitness curve actually starts to descend in the end when the scenarios are not used for evaluation. This indicates that if the GA is not aware of scenarios, it may start to favor solutions which work on a more general level only.

Overall, the fitness curves show that adding the scenario fitness function indeed aids the GA to find more intricate solutions that consider detailed parts of the system.

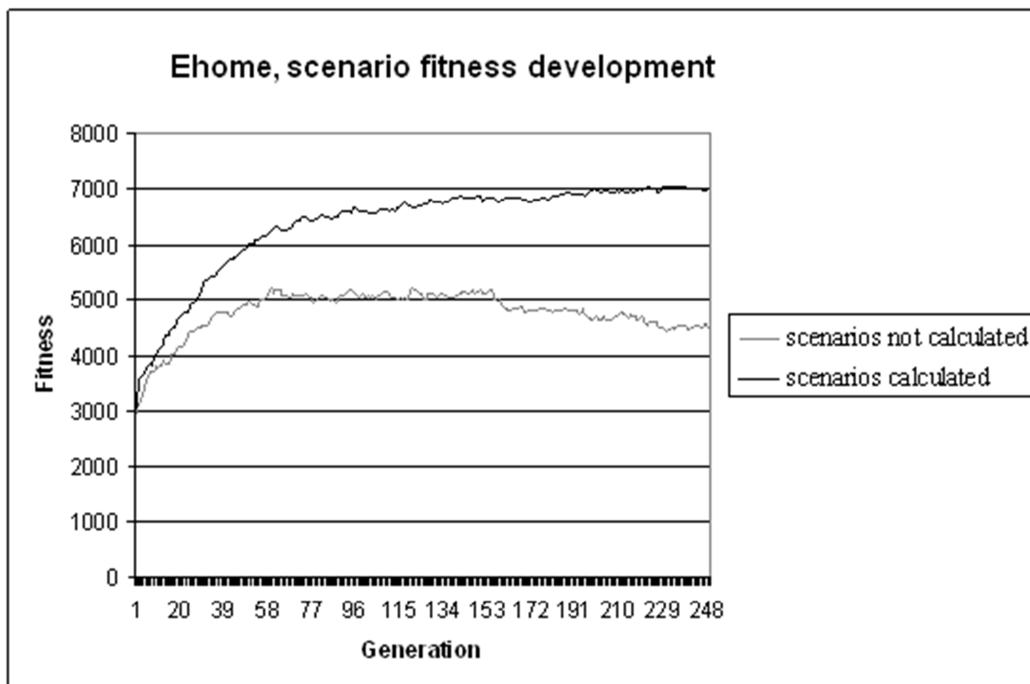


Figure 37. Scenario fitness curves for ehome

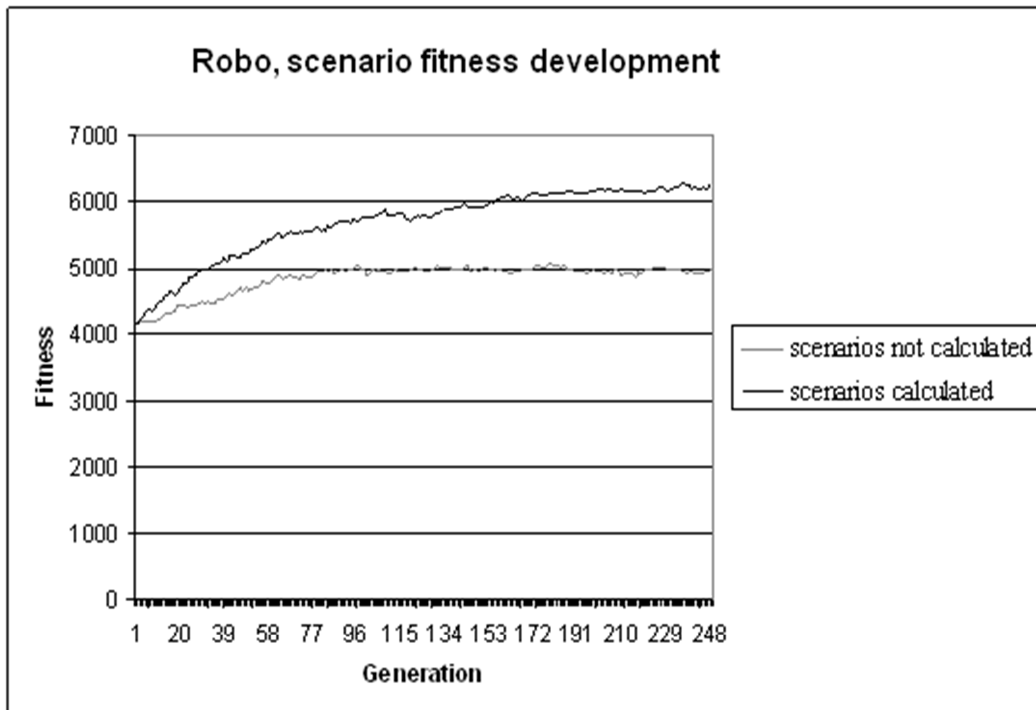


Figure 38. Scenario fitness curves for robo

Once using the scenario fitness had been validated in the case where all sub-fitnesses were given an equal weight, the true affect of scenario-based evaluation could be studied. As the scenarios only deal with modifiability, experiments were also made where the modifiability and scenario sub-fitnesses where given significantly larger weights than efficiency. This enabled studying what kind of solutions the GA would be able to produce when it was mainly guided by scenario evaluation. Example solutions from such scenario enhancing experiments for ehome and robo are given in Figures 39 and 40, respectively.

The example solution for ehome (Figure 39) was achieved from an experiment where the scenario and general modifiability sub-fitnesses were significantly overweighted in relation to efficiency and complexity. As a result, the message dispatcher is present and quite well used (courtesy of overweighting general modifiability) and there are several instances of different low-level design patterns (somewhat due to overweighting scenarios). For example, the Adapter patterns for Heater Manager and Water Manager respond to given scenarios, as these components interface with changeable hardware, and thus the semantics are likely to change. Similarly, showing the music list and coffee machine status have been dealt with Strategies, which is a preferred design solution for the related scenarios.

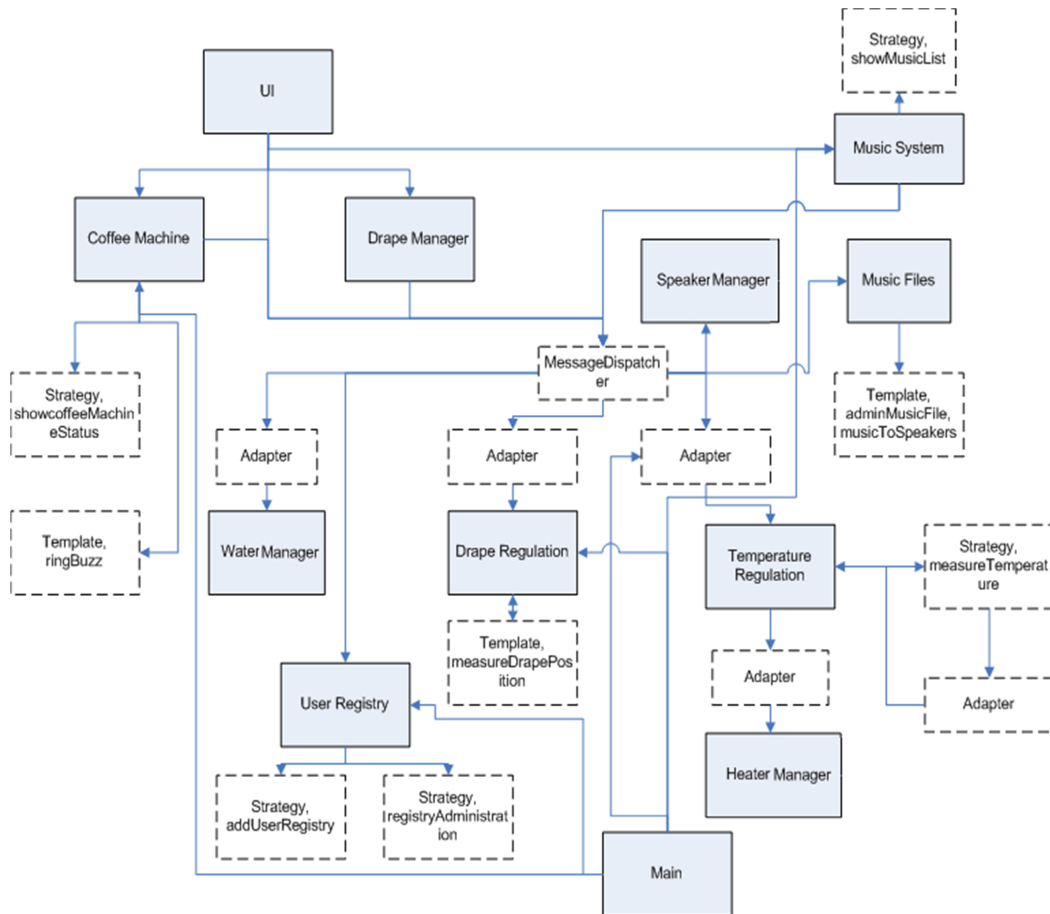


Figure 39. Example solution for ehome when scenarios included and overweighted

The example solution given for robo is not, contrary to other case studies, made with the same parameters as the one for ehome. In the example solution for robo the scenario sub-fitness function has completely replaced the basic modifiability sub-fitnesses. Thus, positive modifiability and negative modifiability were both neglected, and scenarios were heavily overweighted in comparison to efficiency and complexity. This setup serves two purposes: 1) it allows us to see whether scenarios themselves are sufficient in evaluating the architecture as a whole, and 2) the versatility of the algorithm can be evaluated. The second point relates to the definition of the scenario sub-fitness function, where patterns are ranked according to how well they support a certain type of scenario. This may raise the concern that this dictates the direction of design too much, i.e., the design becomes rather deterministic and GA may not use its full potential.

The example solution for robo, given in Figure 40, shows that when scenarios are used instead of the general modifiability fitness and are heavily overweighted, the result is still quite sensible, and thus scenarios can be used for evaluation. However, the amount of patterns is rather small, and thus a larger number of scenarios would be needed if the scenario sub-fitness was used as the only evaluator for modifiability.

Also, the versatility of the genetic algorithm is clear. For example, the scenario for Armor support states that it is highly probable that the developer would want to add an alternative way to control the armor. Thus, a Template Method or Strategy pattern would seem natural ways to handle the situation. However, as seen in Figure 40, the GA has chosen to use Adapter instead. Since an Adapter allows the changing of the interface as well, this solution provides even stronger support for modifications than directly required by the scenarios, with fairly small cost regarding the overall complexity of the system.

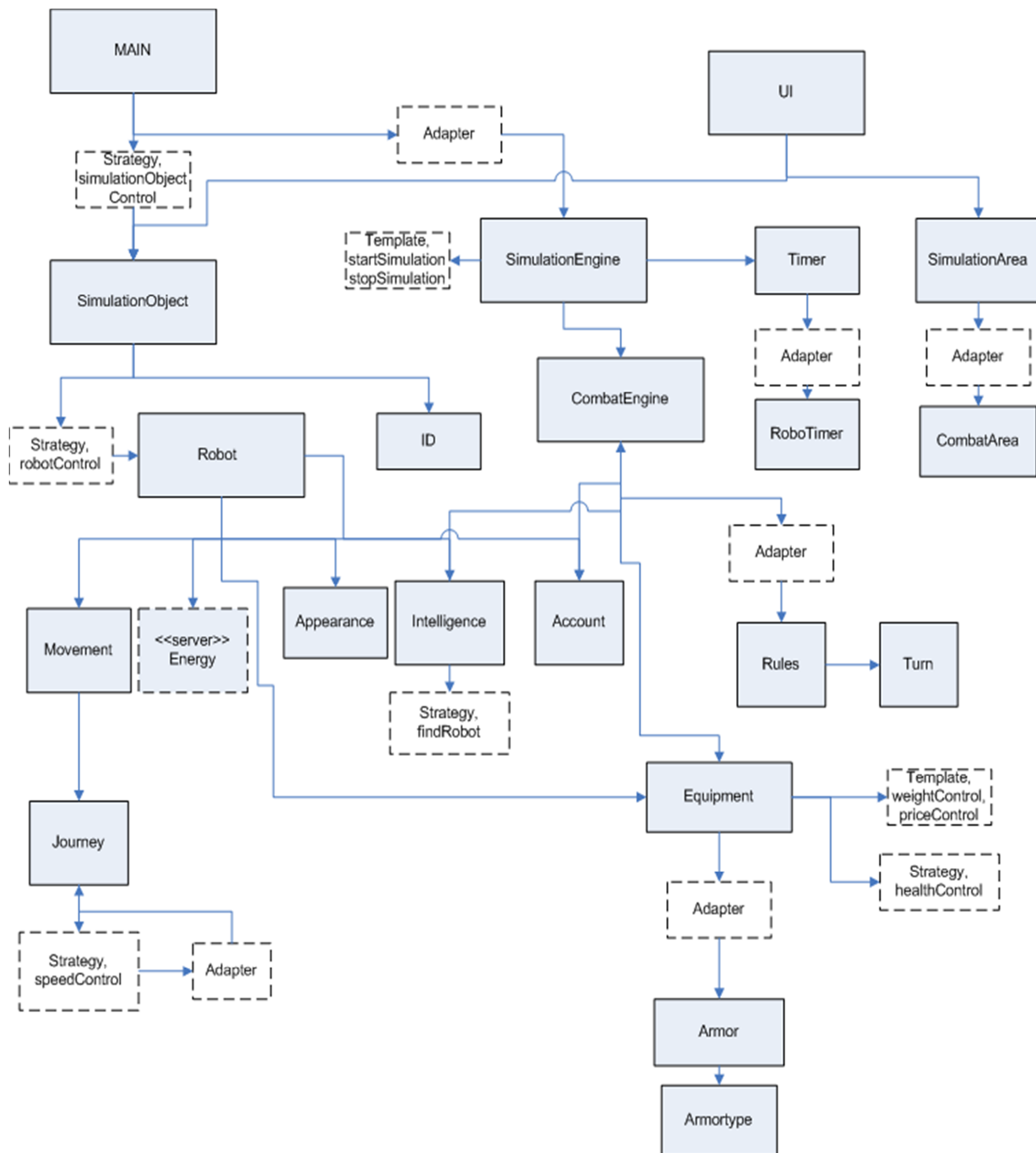


Figure 40. Example solution for robo when scenarios overweighted and replacing general modifiability

The robo example shows well how scenarios may also be supported indirectly. For example, a scenario related to the Intelligence component states that the intelligenceControl operation should be easily changed by the developer; this would be best handled with a Template Method.

However, there is no method within the Intelligence component that uses the intelligenceControl operation, and thus the Template Method cannot be implemented there, as according to the preconditions of the mutation. However, as can be seen in Figure 40, the findRobot operation has been placed behind a Strategy pattern in the Intelligence component. Thus, the GA has chosen an alternative way to satisfy the scenario: as findRobot is used by intelligenceControl, changeability is achieved by separating findRobot behind a pattern, and thus making the intelligenceControl at least partially more modifiable. In many solutions the situation was actually handled similarly: the operations used by the "main" operation of a component were placed behind a Strategy or a Template instead of the "main" operation.

To summarize, modifiability-related scenario-based evaluation is enabled by categorizing scenarios and encoding them for a scenario interpreter, implemented as an extension to the GA. The interpreter provides a preference list of design solutions, which the GA uses to evaluate how well detailed design sub-problems have been handled within the architecture. Results indicate that the scenario sub-fitness enables more detailed evaluation and can be used to enhance the modifiability aspect of designs. This extension to the basic method is discussed in more detail in publication [III].

4.4 MULTI-OBJECTIVITY

No matter how detailed the calculation of different quality attributes is, combining two attributes into one function is still a bit like summing apples and oranges. In other words, while the fitness value may be accurate, it is not very informative unless the distribution of the different quality attributes is known. A solution which receives 1000 points for efficiency and 0 for modifiability is quite different to one that receives 500 points for both properties – just like having 10 apples is quite a different case to having five apples and five oranges, even though in both cases the ultimate value, be it fitness or the number of fruits, is still the same. Thus, the most natural way to handle conflicting quality attributes, such as modifiability and efficiency, is to use multi-objectivity. This enables the evaluation of individuals from both viewpoints separately, and selection of individuals is made based on these individual values, not the combined sum value.

4.4.1 Method

Multi-objective software architecture synthesis is possible through application of Pareto optimality [Deb, 1999]. Pareto optimal selection

evaluates individuals from all viewpoints, and rather than providing one "optimal" individual, a range of satisfactory individuals is presented.

If solutions are measured according to *n* properties, a solution x can be described by a vector $x = [f_1(x), f_2(x), \ldots, f_n(x)]$, where $f_i(x)$ is the value of *i*th property in x. For convenience, it can be assumed that all properties should be maximized. When the properties represent conflicting quality criteria, it is unlikely to find a solution in the solution space *S* which would be optimal with respect to all the quality criteria. In such a situation, Pareto optimality provides a way to compare the solutions.

We say that a solution $x' \in S$ is Pareto optimal if for each $x \in S$, we have either $f_i(x) = f_i(x')$, for all $i = 1, \ldots, n$, or there is at least one property *i* such that $f_i(x) < f_i(x')$. That is, x′ is Pareto optimal if there exists no feasible solution x that increases some criterion without causing a simultaneous decrease in at least one other criterion. Typically, there is not a single solution that is Pareto optimal, but a set of Pareto optimal solutions. The Pareto optimal solutions of a set of feasible solutions are said to form a Pareto front.

In order to apply Pareto optimal selection, each quality attribute needs to be evaluated separately. In order to be able to view results in a two-dimensional graph, this study only concentrates on two quality properties, modifiability and efficiency. The fitness for each solution is thus given in the form $f(x) = [sf_1(x), sf_2(x), sf_3(x), sf_4(x)]$ (using the notation for sub-fitnesses as given in Subsection 3.1.4). For evaluation, the complete values of both quality attributes are used. The complete modifiability value *mf* for solution x is defined as $mf(x) = sf_1(x) - sf_2(x)$ and the complete efficiency value *ef* respectively as $ef(x) = sf_3(x) - sf_4(x)$. Each individual thus has two fitness values, *mf* and *ef*, instead of just one fitness value where the two quality attributes are summed.

Once the fitness values have been calculated, the individuals are ranked based on modifiability (efficiency could be used as well). The Pareto front is then selected so that the most modifiable individual is naturally included, as it represents one of the extreme choices; this can be given the identifier pf_1. The ranked individuals are then processed so that if the potential front individual pf_n has an efficiency value $ef(pf_n) > ef(pf_{n-1})$, it is selected for the front (note, that $mf_n < mf_{n-1}$, as the individuals are ranked in descending order according to modifiability values).

As the population contains 100 individuals, the Pareto fronts obtained with the presented approach usually contain approximately five to ten individuals. Hence, when selecting a population, one front is not enough for the genetic algorithm to progress. Thus, in my approach, when the mutations and crossovers for population p_n have been performed, the Pareto front of p_n is selected for the next generation (population p_{n+1}).

After this, the Pareto front of the remaining individuals, i.e., the Pareto front of population $p_n \setminus p_{n+1}$, is selected and transferred to the next generation. This iterative process is repeated until the population p_{n+1} has at least 100 individuals.

Once the evolution process is complete, the outcome is not just one class diagram, but the class diagrams (synthesized architectures) of the entire Pareto front of the final population. This enables the architect to view the different extremes without having to manipulate weights for the fitness function.

4.4.2 Case studies

The case studies were used to study two different aspects: 1) how do the Pareto fronts evolve during the evolution, and 2) what kind of solutions are produced in a front, especially how much does the most efficient solution differ from the most modifiable solution.

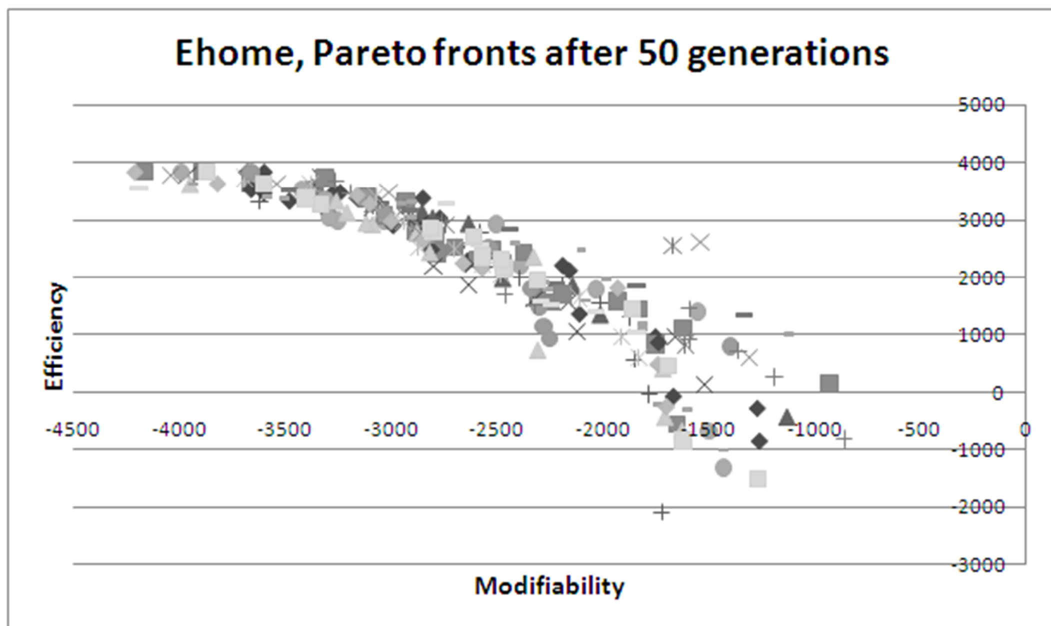


Figure 41. Initial Pareto fronts for ehome

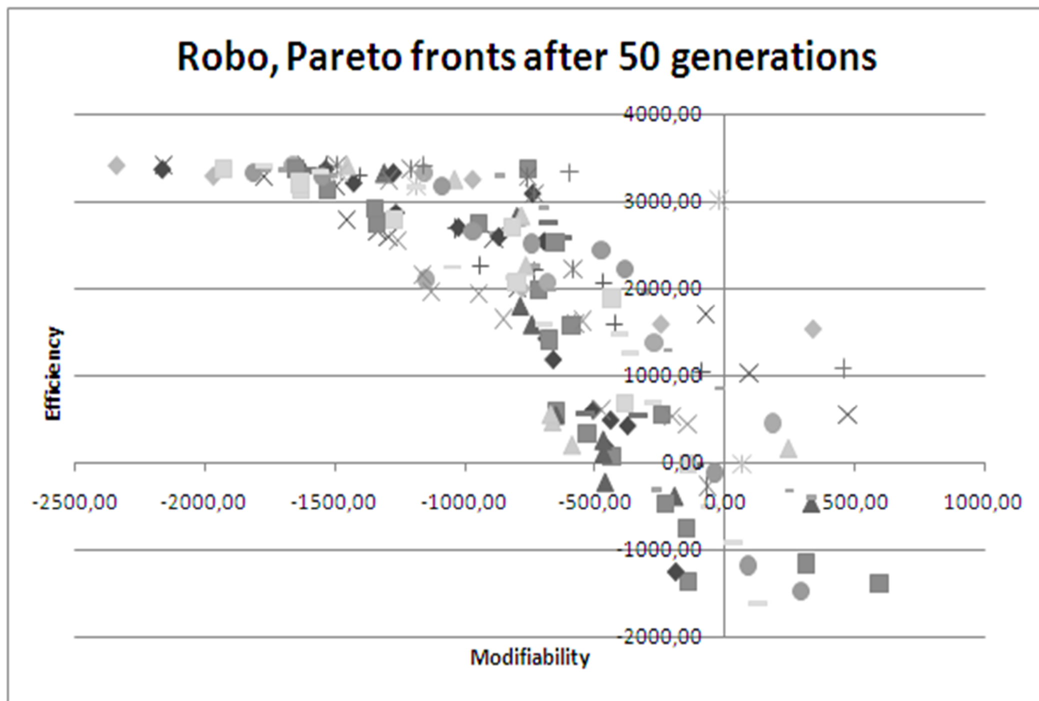


Figure 42. Initial Pareto fronts for robo

Firstly, the Pareto fronts of both cases were studied. Figure 41 shows the Pareto fronts of 20 runs for ehome in the beginning of evolution (after 50 generations), and Figure 42 shows the respective Pareto fronts for robo. Each Pareto front (different run) is represented by a distinctive marker. As can be seen, in the begininning the fronts are quite clustered in the efficient section of the graph, and in Figure 41 (ehome) no individual actually has a positive modifiability value at this point. In Figure 42 (robo) there are some individuals which have positive modifiability, but they are few, and the trend is the same as with ehome, as individuals are quite clustered and focused on the efficient section.

Figures 43 and 44 show how the Pareto fronts have evolved as the figures portray the final Pareto fronts at the end of evolution for ehome and robo, respectively. The Pareto fronts have moved towards the modifiable section of the graph, and the efficient solutions are still quite tightly clustered, while within the modifiable solutions there is much more variance. This is expected, as efficient solutions are close to the null architecture, i.e., have as few mutations as possible, while good modifiability values can be reached with a practically endless number of pattern combinations.

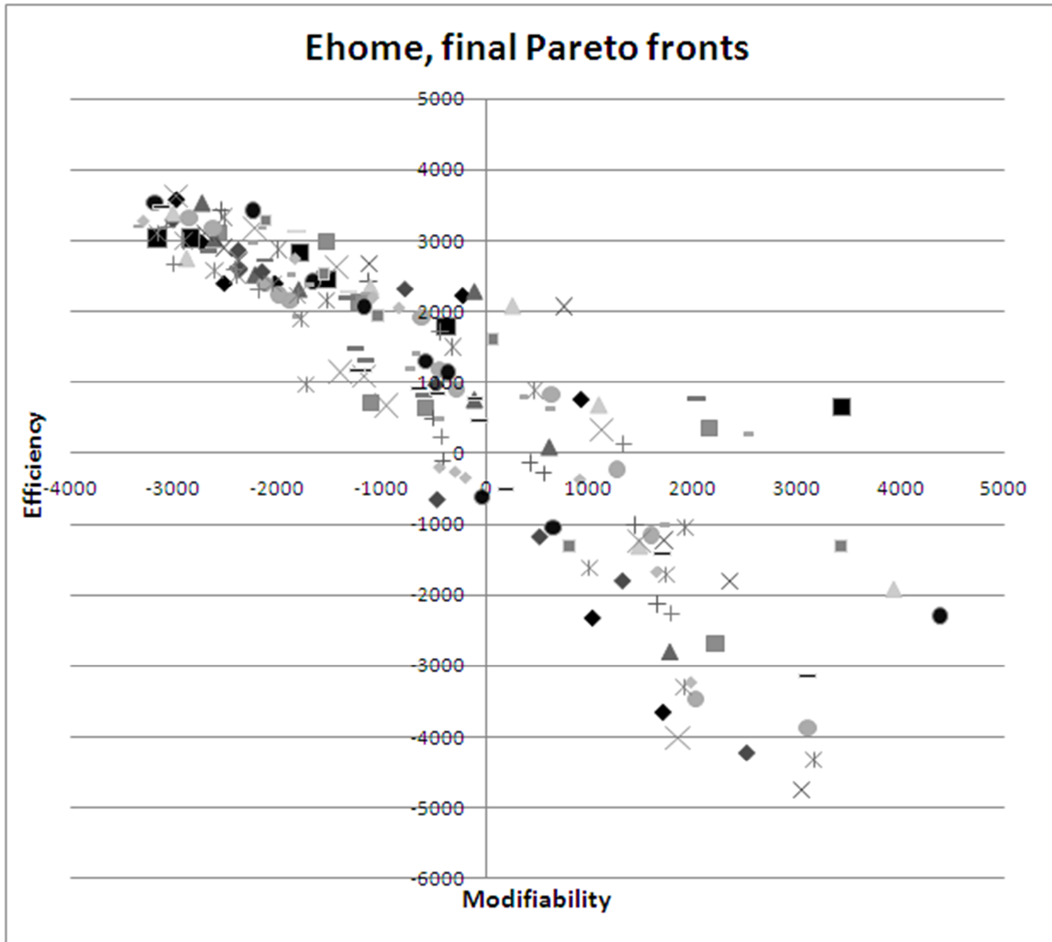


Figure 43. Final Pareto fronts for ehome

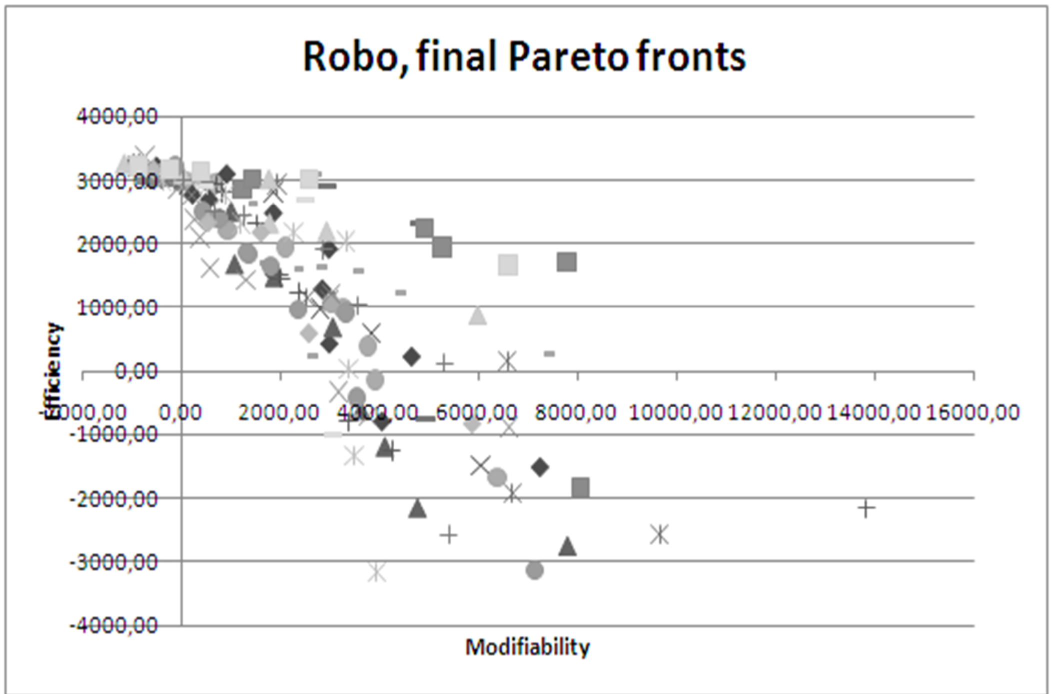


Figure 44. Final Pareto fronts for robo

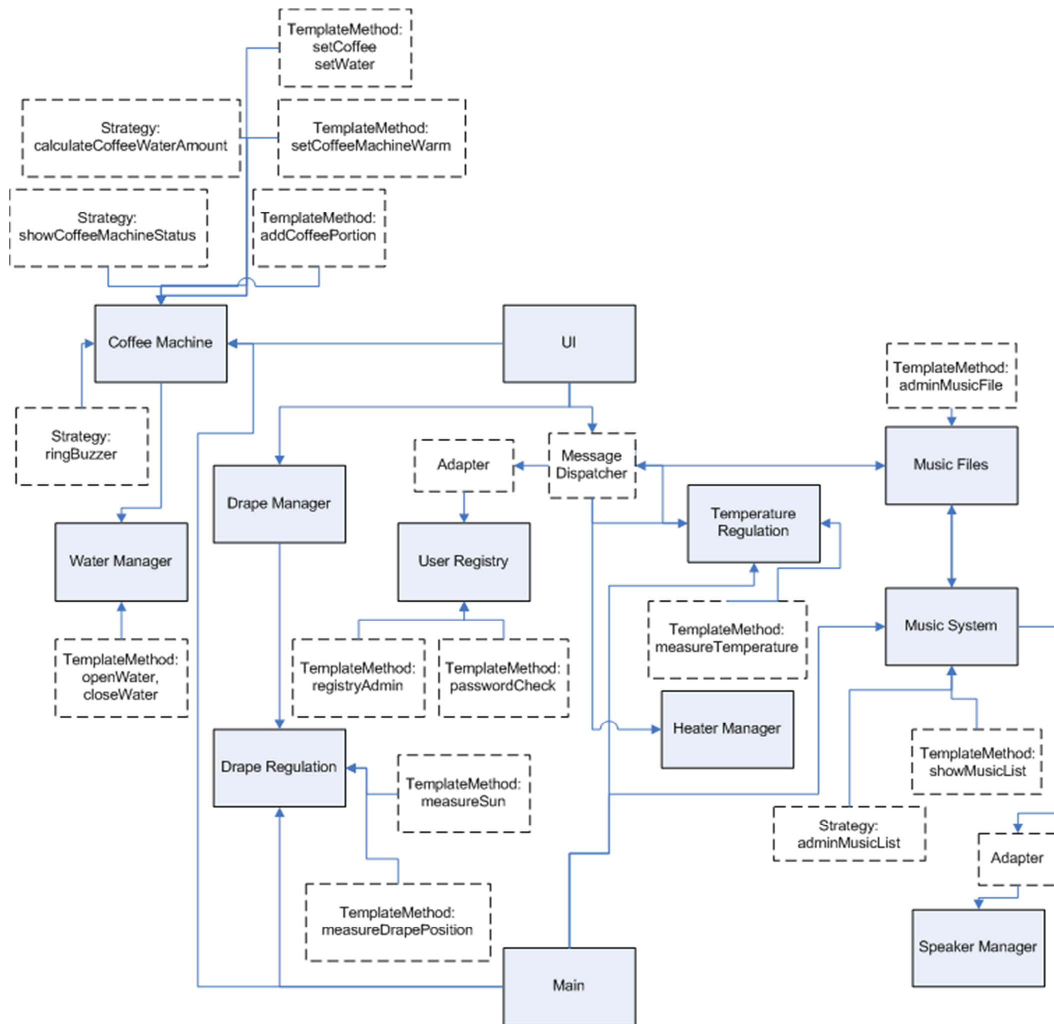


Figure 45. Example solution for ehome from modifiable end of Pareto front

Secondly, the example solutions for both case studies where inspected. The final Pareto front of one of the test runs was selected for exemplary purposes for both ehome and robo. The most modifiable solution from this exemplary Pareto front for ehome is illustrated in Figure 45, and the most efficient solution from the same Pareto front is given in Figure 46. The difference between these two solutions is substantial. In the modifiable solution the message dispatcher style is central, and there are several instances of all low-level design patterns. On the other hand, in the efficient solution the message dispatcher is not present, the number of design patterns is much smaller. Also, in the efficient solution, most of the design patterns are instances of Template Method, which has the least effect on efficiency. This represents quite a typical front for ehome, although several fronts also had the message dispatcher in the most efficient solution as well, but it only had one or two connections, so it was very poorly used.

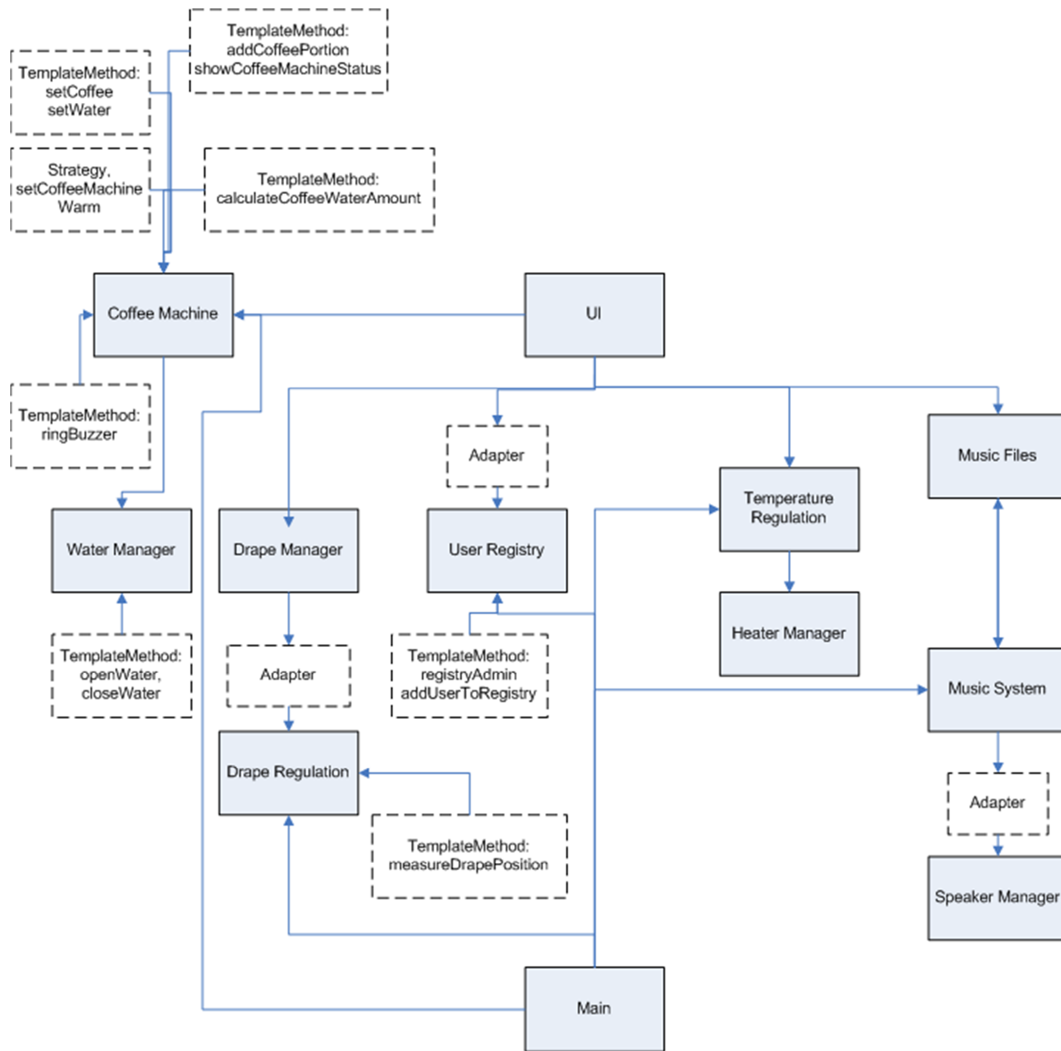


Figure 46. Example solution for ehome from efficient end of Pareto front

As for robo, the most modifiable solution of the exemplary Pareto front is portrayed in Figure 47, while the most efficient individual from that same front is given in Figure 48. The solutions are very similar to those achieved with ehome: in the modifiable solution the message dispatcher is heavily used, and there are several instances of all low-level design patterns. In the efficient solution there are no Strategy patterns, which give the highest penalty in terms of efficiency, and the message dispatcher is not used at all.

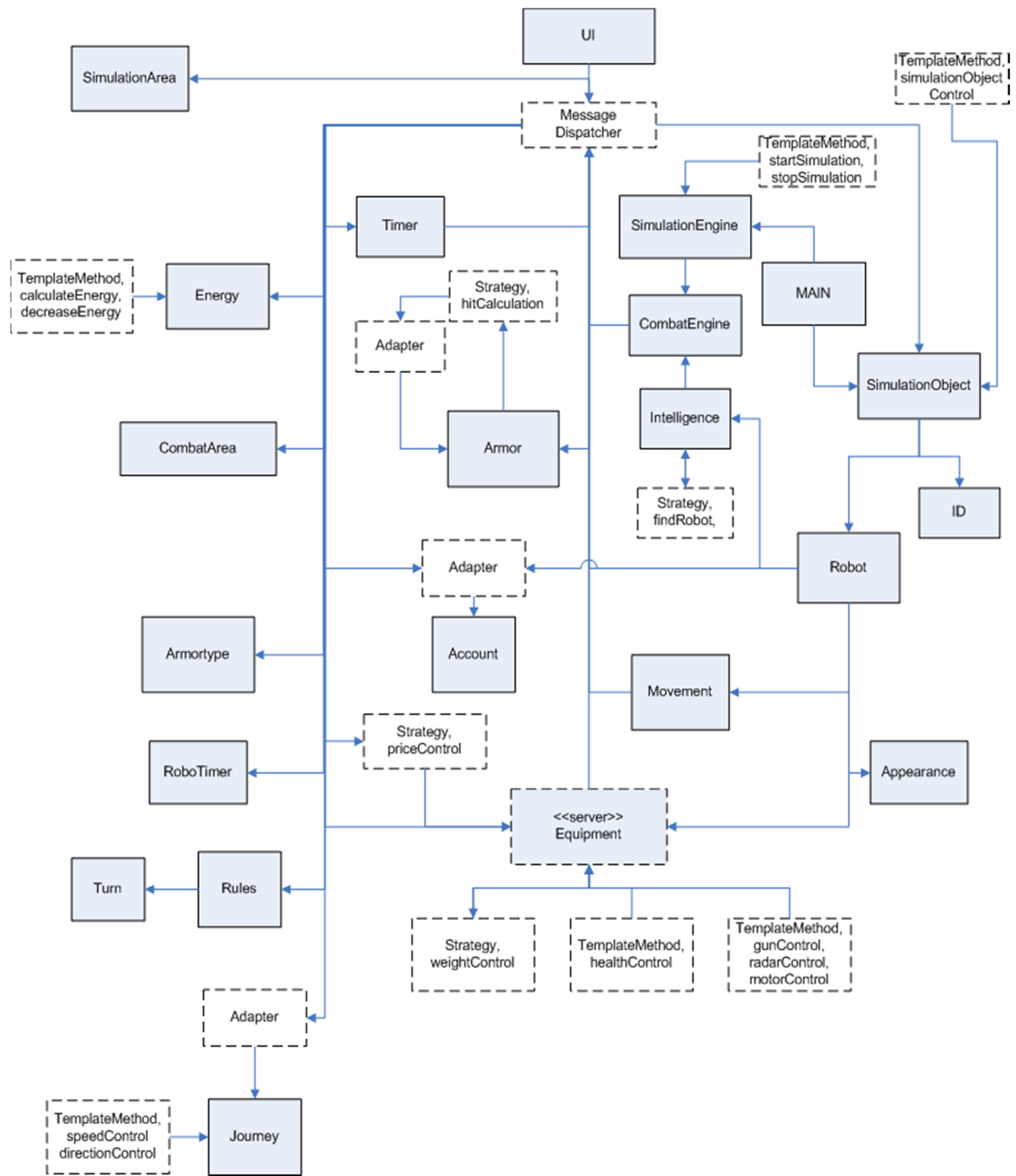


Figure 47. Example solution for robo from modifiable end of Pareto front

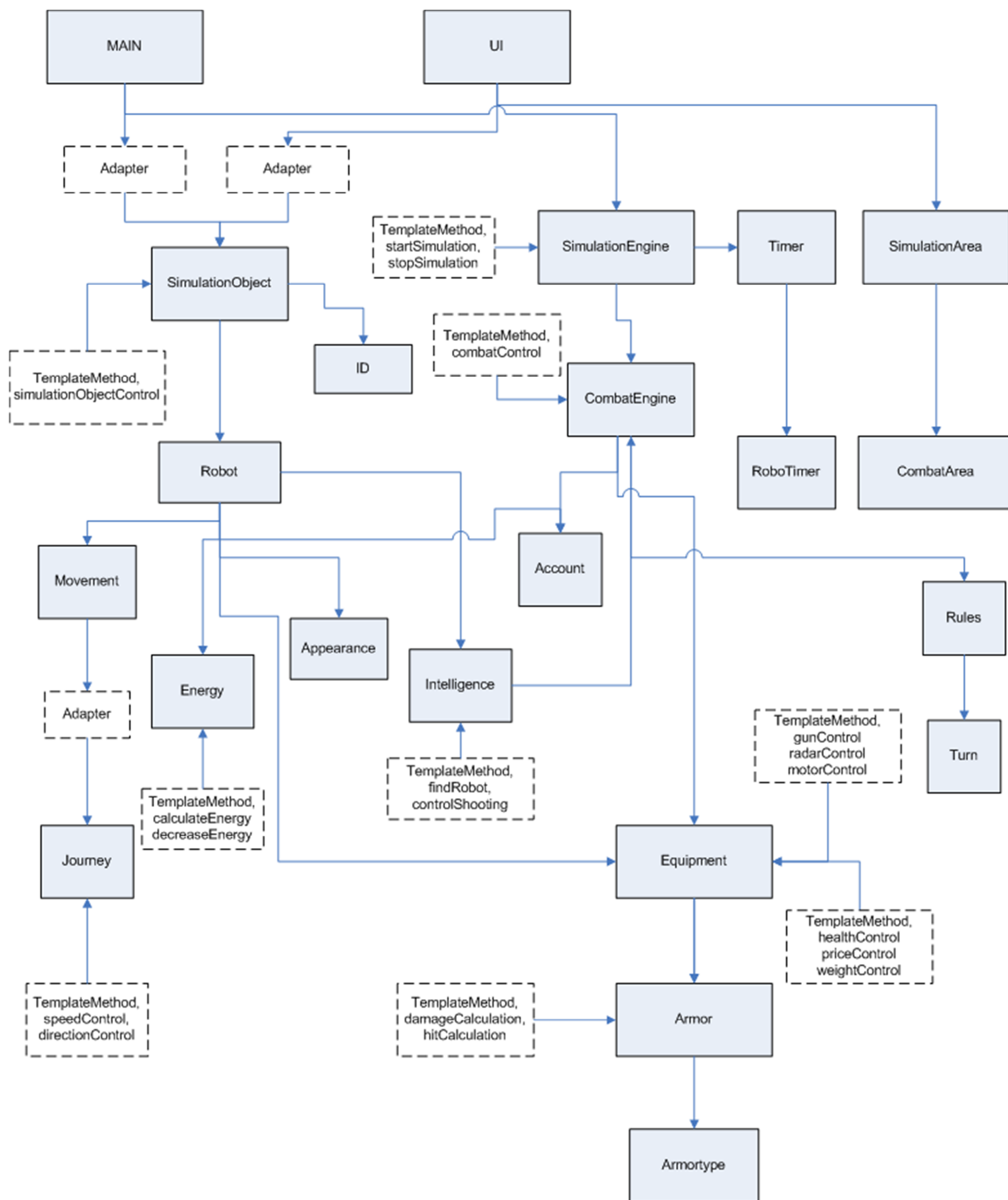


Figure 48. Example solution for robo from efficient end of Pareto front

To summarize, the multi-objective approach, implemented using the Pareto optimality concept, provides a way to evaluate each solution from different viewpoints. It also provides several candidate solutions instead of only one. This equips the designer with two kinds of information: 1) by examining the solution and fitness distributions it is easier to see how a chosen quality attribute affects the outcome of the design and 2), by seeing several solutions at once, using one solution as a starting point for the actual design can be done with more confidence, as opposed to using the one solution produced by a single weighted function, when it is not clear why the algorithm suggests that one particular solution.

Results from case studies show that the extremes of Pareto fronts are very different, and thus the entire front gives a complete view of what kinds of solutions the algorithm is able to produce. The multi-objective approach is discussed in publication [IX].

5 Evaluation

While the previous chapters presented the method for GA-based software architecture synthesis and its different variations, the results still need validation. In this chapter, I will begin by summarizing the results – how different variations affect the outcome and what are the differences between the case studies. I will then present evaluation for the multi-objective approach by examining the individuals of a Pareto front against ATAM-type scenarios. After this, an experimental study is presented to evaluate the results obtained by the (basic) GA approach by comparing the synthesized solutions with human-made solutions. Finally, the different factors affecting the outcome of case studies and the experimental study are discussed, as well as different aspects related to the synthesis that were discovered during the research process but not covered in the publications or case studies.

5.1 Summary of Results

In order to provide a compact comparison of the effect of different variations to the basic method, the results from case studies with different variations are summarized. In Chapter 3, the basic method and case studies were presented. The basic synthesizer uses a single weighted fitness function, and thus the main issue becomes scaling the weights so that all quality attributes are equally appreciated, as by default the architecture should satisfy as many quality requirements as possible. Thus, the resulting solution proposals support all three attributes to some extent. The solutions for both ehome and robo are not especially good from any particular quality viewpoint, but they are not particularly bad, either. While these designs are acceptable, i.e., they show that automation is possible, they are far from the quality of a "final" design, and much

revision would be required from the architect. Also, as only a single fitness value and a single solution is provided after each evolution, it would require much knowledge from the user of how the algorithm should be tuned in order to achieve better results.

The case studies for asexual reproduction, discussed in Section 4.1, showed that asexual reproduction can be used to speed up the evolution, as it is significantly faster than the standard version with random crossover, but that the GA tended to converge to a local optimum. The resulting solutions were quite focused on only a couple of low-level design choices, and there was little variation between results. The same kind of results have actually been achieved with simulated annealing [Räihä et al., 2010], which also operates with mutations only. The main difference is that the solutions with simulated annealing were heavily relying on the message dispatcher, while with asexual reproduction and the GA they relied on low-level patterns. The trend is, however, the same: only few design options are used to refine the solution.

Complementary crossover, as presented in Section 4.2, produced much better results, as both the fitness values and produced solutions achieved quite a different level of quality than any tests until then. Purposefully combining parents seemed to enable solutions with delayed reward, which resulted in solutions concentrating around the message dispatcher, which has a big effect on both modifiability and efficiency. The solutions also utilized the low-level patterns in addition to the message dispatcher.

In Section 4.3 the scenario-based extension to the core fitness function was given. The purpose of using scenarios was to enable more detailed fitness evaluation by drawing attention to the specific needs of individual operations. Using the scenario fitness function as an extension to the core fitness or as a substitute for the basic modifiability fitness produced solutions where the patterns were better targeted to the operations specified in the scenarios. However, scenarios did not appear well-suited to make higher-level design decisions for the architecture.

As mentioned earlier, there are many problems which come from using a single weighted fitness function. Thus, a multi-objective variation using the Pareto optimality concept was discussed in Section 4.4. This effectively solves the problem of only having average individuals, as a set of individuals is always produced, and the set contains solutions from both extremes of the modifiability-efficiency space. No weights are needed for this approach, although they were used in the experiment to reduce misconceptions due to scaling. When examining the Pareto optimal solutions, the difference between the two extreme ends is evident: the modifiable solutions typically employed the message dispatcher as well as

several low-level design patterns, while the efficient solutions contained a few low-level patterns only, usually those with least effect on efficiency.

When comparing the two case studies, robo seems “easier” to deal with from the GA point of view. It has many small classes, but most still have more than one functional operation. Thus, there are many options to introduce both low-level and high-level solutions. On the other hand, in the case of ehome, there are much fewer classes, which makes the design slower especially in terms of low-level patterns. The “maximum” number of simultaneous patterns is met much faster than with robo, and thus the old patterns need to be removed before new, better-suited ones can be introduced. Also, fewer connections between classes means less opportunities to apply the message dispatcher, which makes it difficult to retain the dispatcher in the architecture, if delayed reward is not supported. The solutions for both case studies were quite similar in terms of how low-level patterns were used, and the main difference was the use of the message dispatcher style.

Thus, to summarize, much variation in solutions could be seen when conducting within-case analysis between different variations to the synthesis. However, analysis between cases revealed only small differences in solutions, the main difference being the level of message dispatcher usage.

5.2 Evaluation of Pareto Fronts

In addition to the traditional case studies, an empirical study was executed with the multi-objective approach in order to investigate whether the Pareto front corresponds to the idea that a human software engineer has regarding what is modifiable and where should efficiency be emphasized. The solutions from five randomly chosen Pareto fronts for ehome were evaluated against scenarios (simulating an ATAM-like evaluation) in order to see whether the Pareto fronts correspond to real-life distribution in the modifiability-efficiency space. Use cases were used to construct five efficiency related scenarios, one for each major subsystem (coffee machine, user registry, drape control, temperature control and music system). Each scenario gives efficiency penalty *ep* to solution x according to the following formula:

$$ep(x) = -\sum \text{calls between different classes} \\ - d*\sum \text{calls via dispatcher} \\ - s*\sum \text{calls to server},$$

where d and s are cost factors of message-based and client-server communications, respectively (relative to straight calls). In these experiments both d and s were set to 2.

For evaluating modifiability, software engineering experts not involved in this study were asked to construct 4-5 modifiability related scenarios for ehome, in a similar fashion as described in Section 4.3. The experts produced a total of 12 modifiability related scenarios, which covered each of the major subsystems. A solution was evaluated against a scenario by awarding 0, 1, or 2 points in the following fashion:

0 points: existing code would have to be changed (no support),

1 point: existing code need not be changed, but the architecture supports development time variation rather than run-time variation (partial support),

2 points: existing code need not be changed (full support).

The points were then multiplied with the probability of the scenario (a real value between [0, 1]) to achieve the total modifiability reward value for an architecture.

Every solution of each of the five fronts was evaluated against each modifiability and efficiency scenario, and the scenario points were compared against how the solutions were distributed in the Pareto front. In particular, if a solution received the least efficiency penalty from scenarios (i.e., it is highly efficient), it should also be the most efficient solution of the Pareto front. Similarly, the solution which received the highest modifiability points from the scenarios should be the most modifiable solution of the Pareto front.

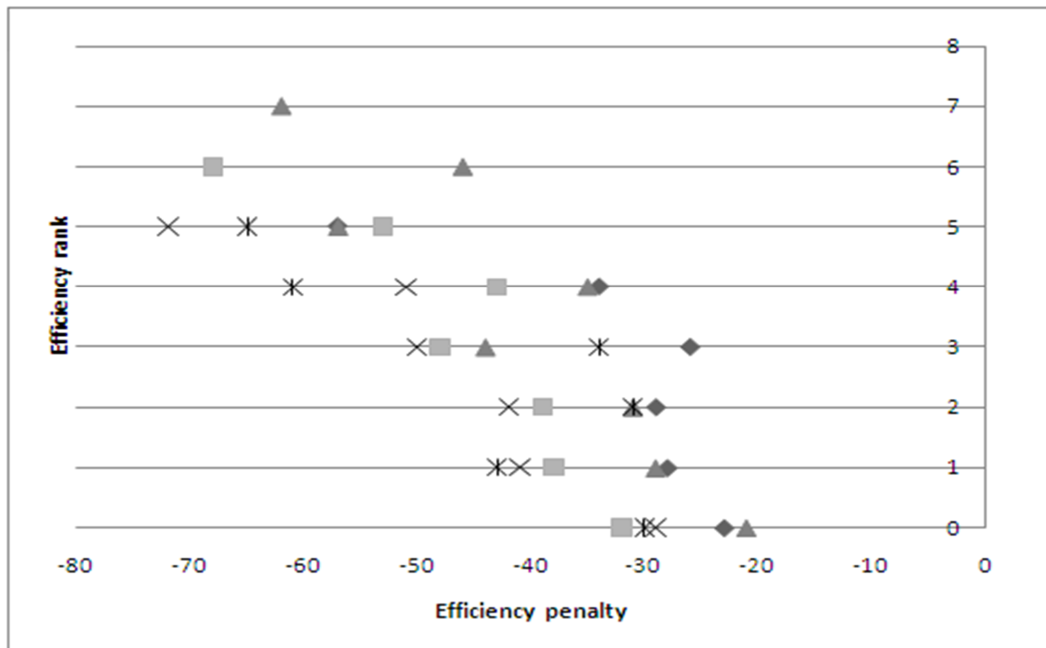


Figure 49. Efficiency scenarios

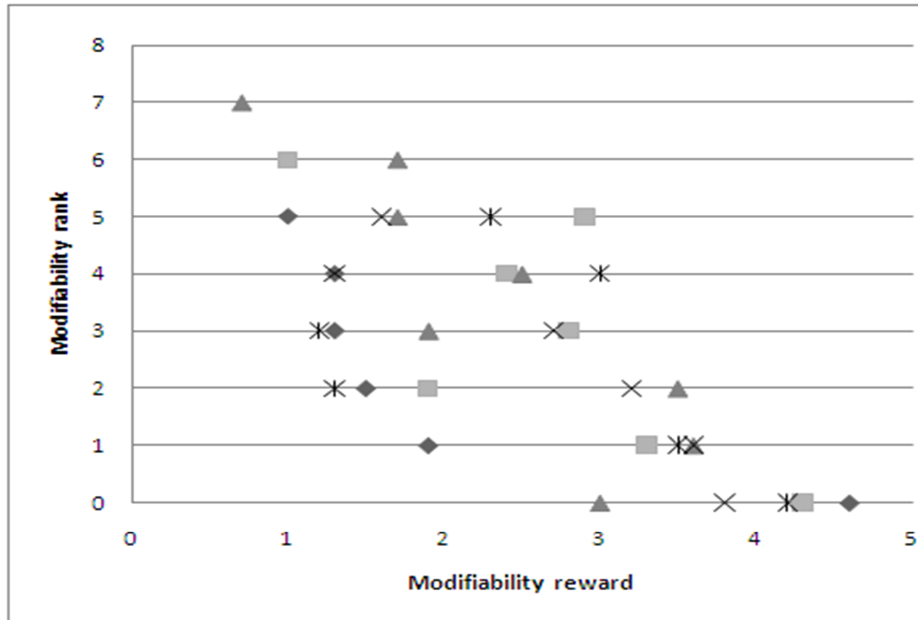


Figure 50. Modifiability scenarios

Figure 49 shows the scatter plot for efficiency. Each Pareto front has a distinctive marker in the plot. The y-value is the efficiency rank, i.e., number 0 is the first, and thus the most efficient, individual of the front (considering fitness values). Thus, the lower a marker, the closer to zero (small penalty) it should be on the x-axis. As can be seen, the solutions with foremost ranks perform the best and those in the last ranks perform the worst in the scenarios as well, while in the middle there is some variation. This is quite logical and expected, as the differences in these kinds of "in-between" solutions are usually quite small. They often have moderate usage of design patterns, and may use the message dispatcher to some extent, but usually there are no distinct differences so that one "in-between" solution would be intuitively better in terms of some quality attribute than another "in-between" solution.

Figure 50 gives a similar scatter plot for the modifiability scenarios. The markers here are the same as in Figure 49, i.e., the front represented by triangles in Figure 49 is represented by triangles in Figure 50 as well. Again, the individual with rank 0 is the most modifiable one of a given front, so the lower a mark, the higher its x-value (modifiability reward) should be. The relationship between high modifiability in fitness values and high scenario reward is clear, although the result is not quite as good as with efficiency, as there is significantly more variance in the solutions in the middle of the Pareto fronts. This is natural, as the scenarios do not capture the particularities of the entire system, but merely attempt to cover the modifiability requirements with examples, which may have the drawback of producing inconsitent results. This drawback is emphasized in the case of modifiability.

Performing statistical analysis on the scenario data produces correlation coefficient r values r = 0.79 for efficiency and r = -0.62 for modifiability. This indicates a high linear correlation between efficiency scenario points and the Pareto front rank, and a moderate correlation between the modifiability scenario points and the Pareto front rank. The corresponding coefficients of determination R^2 values are $R^2 = 0.62$ for efficiency and $R^2 = 0.52$ for modifiability, meaning that 62% of the variation in efficiency scenario penalty points can be explained by the Pareto rank, and similarly, 52% of variance within modifiability scenario reward points can be explained by the Pareto rank. Finally, the Student's t-tests for both data sets show that the impact of the rank in scenario points is significant with a 95% confidence (i.e., there is a significant correlation between rank and scenario points with $p < 0.05$), and thus, the scenario points depend on the Pareto rank (i.e., the higher the rank, the better it performs in the scenarios). This experiment is discussed in publication [IX].

5.3 EMPIRICAL STUDY

As shown in the case studies, genetic software architecture synthesis is able to produce reasonable architecture proposals, although obviously they still need some human polishing. However, fitness values do not have a straightforward correlation with expert evaluations of "good" architectures. Thus, in order to answer the final and ultimate research question of how far the synthesis can be taken, an empirical study was conducted. In this experiment the quality of the generated architectures is studied in relation to the quality of architectures produced by students. This empirical study is further discussed in publication [VIII].

5.3.1 Setup

First, a group of 38 students from an undergraduate software engineering class was asked to produce an architecture design for ehome. Most of the students were third year Software Systems majors from Tampere University of Technology, having participated in a course on software architectures.

The students were given essentially the same information that is used as input for the GA, i.e., the null architecture, the scenarios, and information about the expected frequencies of operations and their expected sensitiveness to variation. In addition, students were given a brief explanation of the purpose and functionality of the system. They were asked to design the architecture for the system using only the same architecture styles and design patterns that were available for GA. The students were instructed to consider efficiency, modifiability and complexity in their designs, with an emphasis on modifiability. It took 90 minutes on average for the students to produce a design.

The synthesized solutions, in turn, were achieved in 38 runs, resulting in 38 architecture proposals. Each run took approximately one minute (i.e., it took one minute for the synthesizer to produce one solution).

The assistant teacher for the course (impartial to the GA research) graded the student designs as test answers on a scale of 1 to 5, 5 being the highest. The solutions were then categorized according to the points they achieved, and one solution from each of the categories of 1, 3 and 5 was randomly selected. These architectures were presented as grading examples to four software engineering experts. The experts were researchers and teachers at the Department of Software Systems, Tampere University of Technology. They all had a M.Sc. or a Ph.D. degree in Software Systems or in a closely related discipline and several years of expertise from software architectures, gained by research or teaching. They were given the same information as the students regarding the requirements for ehome.

In the actual experiment, the experts were given 10 pairs of architectures. One solution in each pair was a student solution, selected randomly from the whole set of student solutions, and one was a synthesized solution, also selected randomly. The solutions were edited in such a way that it was not possible for the experts to know which solutions were synthesized, while still keeping all information of architectural design decisions. The experts were not told how the solutions were achieved, i.e., that they were a combination of student and synthesized solutions. They were merely asked to help in evaluating how good solutions a synthesizer could make. The experts were then asked to give each solution 1, 3 or 5 points. The setup is discussed in more detail in publication [VIII].

5.3.2 Results

The scores given by the experts (e_1-e_4) both to the automatically synthesized architectures (a_1-a_{10}) and architectures produced manually by the students (m_1-m_{10}) are given in Table 1. As the experts viewed the solutions pair-wise, the points in Table 1 are also organized in pairs of synthesized and manually produced solutions. The result of each comparison within a solution pair is one of the following

- the synthesized solution is considered better ($a_i > m_i$, denoted later by +)
- the student solution is considered better ($m_i > a_i$, denoted later by -), or
- the solutions are considered equal ($a_i = m_i$, denoted later by 0).

Table 1. Points for synthesized solutions and solutions produced by the students

	a_1	m_1	a_2	m_2	a_3	m_3	a_4	m_4	a_5	m_5	a_6	m_6	a_7	m_7	a_8	m_8	a_9	m_9	a_{10}	m_{10}
e_1	3	3	1	3	5	3	1	5	3	1	1	3	3	3	5	3	3	5	3	3
e_2	5	1	3	3	5	1	1	1	3	3	3	5	1	1	3	1	1	1	5	1
e_3	3	3	3	5	3	3	1	3	3	1	3	1	1	3	1	1	3	3	3	1
e_4	3	1	5	3	3	5	3	1	5	1	5	3	3	3	3	1	3	3	5	1

The best synthesized solutions appear to be a_3 and a_{10}, with two 3's and two 5's. In solution a_3 the message dispatcher was used, and there were quite few patterns, so the design was easy to understand while still being modifiable. However, a_{10} was quite the opposite: the message dispatcher was not used, and there were especially as many as eight instances of the Strategy pattern, when a_3 had only two. There were also several Template Method and Adapter pattern instances. In this case the solution was highly modifiable, but also quite complex. This demonstrates how very different solutions can be highly valued with the same evaluation criteria, when the criteria are conflicting: it seems impossible to achieve a solution that is at the same time optimally efficient, modifiable and still understandable.

The worst synthesized solution was considered to be a_4, with three 1's and one 3. This solution used the message dispatcher but also the client-server style was eagerly applied. There were not very many patterns, and the ones that existed were quite poorly applied. Among the human-made solutions, there were three solutions (m_5, m_8, and m_{10}) with similar scoring.

Table 2 shows the numbers of the preferences of the experts, with "+" indicating that the synthesized proposal was considered better than the student proposal, "-" indicating the opposite, and "0" indicating a tie. Only one (e_1) of the four experts prefers the student solutions slightly more often than synthesized solution, while two experts (e_2 and e_4) prefer the synthesized solutions clearly more often than the student solutions. The fourth expert (e_3) prefers both types of solutions equally. There were totally 17 pairs of solutions with better score for the synthesized solution, 9 pairs preferring the student solution, and 14 ties.

The presented crude analysis clearly indicates that in this simple experiment, the synthesized solutions were ranked at least as high as student-made solutions. Thus, it can be deduced that synthesized solutions at this stage are competitive with those produced by third year software engineering students.

Table 2. Numbers of preferences of the experts

	+	-	0
e_1	3	4	3
e_2	4	1	5
e_3	3	3	4
e_4	7	1	2
total	17	9	14

5.3.3 Threats and limitations

There are several threats and limitations to the presented experiment. Firstly, as the solutions for evaluations were selected randomly out of all the 38 solutions, it is theoretically possible that the solutions selected for the experiment do not give a true representation of the entire solution group. However, as all experts were able to find solutions they judged worth of 5 points as well as solutions only worth 1 point, and the majority of solutions were given 3 points (i.e., the distribution of points roughly followed the Gaussian normal distribution), it is unlikely that the solutions subjected to evaluation would be so biased it would substantially affect the outcome of the experiment.

Secondly, the pairing of solutions could be questioned. The evaluation could have been more diverse if the experts were given the solutions in different pairs (e.g., for expert e_1 the solution a_1 would have been paired with m_5 instead of m_1). One might also ask if the outcome would be different with different pairing. However, as the overall points are better for the synthesized solutions, different pairing would most probably not significantly change the outcome. Also, the experts were not actually told to evaluate the solutions as pairs – the pairing was simply done in order to ease the evaluation and analysis processes.

Thirdly, the actual evaluations made by the experts should be considered. Naturally, having more experts would have strengthened the results. However, the evaluations were quite uniform. There were very few cases where three experts considered the synthesized solution better or equal to the student solution (or the student solution better or equal to the synthesized one) and the fourth evaluation was completely contradicting. In fact, there were only three cases where such contradiction occurred (pairs 2, 3 and 4), and the contradicting expert was always the same (e_4). Thus, the consensus between experts is sufficiently good, and increasing

the number of evaluations would not substantially alter the outcome of the experiment in its current form.

Finally, the task setup was limited in the sense that architecture design was restricted to a given selection of patterns. Giving such a selection to the students may both improve the designs (as the students know that these patterns are potentially applicable) and worsen the designs (due to overuse of the patterns). Unfortunately, this limitation is due to the genetic synthesizer in its current stage, and could not be avoided.

5.4 Discussion

When the general implementation for GAs was defined in Section 2.2, it was stated that many different variables, e.g., encoding, mutation and crossover operations, and fitness function, need to be carefully considered in order to find satisfying solutions. Naturally, all of these affect the outcome of the search. The effect of these different variables as well as the properties of the different test cases and their effect to the results are now discussed. Finally, the limits and potential of the approach are considered.

5.4.1 Input and encoding

The chosen encoding is operation-centric, which did not affect the outcome in any other way than in terms of performance. When, say, preconditions for certain patterns required a more class-centric perception of the architecture, the operation-centric supergene encoding was not the most efficient data structure for the algorithm. However, the operation-centric approach enabled the architecture to be encoded in such a way that firstly, no operation would ever be "lost", and secondly, mutations were easily targeted. Had the encoding been class-centric, a feasibility check would have been required after each crossover and mutation in order to make sure that each operation (defining the system) still belongs to some class, and is therefore included in the solution. Thus, a separate subfunction for the GA would have been required to perform this check, and to reassign operations to classes if needed. As for mutations, only a single random mutation point selection sufficed to target a pattern to an operation, and there is no need to first perform a class selection (to find the gene where mutation should be targeted) and then another selection for operation (as patterns concentrate on operations, a specific operation should also be selected).

While the selected encoding had no effect on the quality of produced solutions, the actual input given for encoding naturally did. The GA received as input the dependencies (calls) between different operations and an initial class structure (null architecture). Additionally, the

(relational) values for frequency of use, sensitivity to variation and amount of data needed were roughly estimated for each operation.

The null architecture had a major impact on the outcome, as the GA was not allowed to alter the class structure in any other way than by applying patterns. In the very beginning of this research (publication [I]), the class structure was not given as input, and in addition to the pattern mutations, the GA could perform mutations such as splitting a class (dividing one class and the operations within it into two different classes), and merging classes (merging two different classes and the operations within them into one class). The GA was at that point also completely in charge of interfaces, and a separate interface for each operation could be added (or removed). However, it quickly became apparent that trying to simultaneously solve both the CRA problem (which assigning operations to classes essentially is) and a higher level architecture optimization problem was too much. The studies by Simons et al. [2010] and Bowman et al. [2010] also clearly indicate that the CRA problem is very complex and difficult to solve as it is, even without the burden of additional design problems. Thus, the algorithm should either work in two stages – first solve the class responsibility assignment problem and then begin introducing the patterns, or only concentrate on the higher level architecture and patterns. As the main focus in this work was in synthesizing the higher level architecture design decisions, and as a rudimentary class structure could quite deterministically be derived from use case and sequence diagrams, giving a null architecture as input seemed the best choice, although the limits it sets to the synthesis are recognized.

As stated, some properties of operations were also estimated and given as additional input. These estimated values were used as a part of the fitness function in order to have more accurate fitness values. The effect of these estimated values could be seen in the architectures at some level, but they did not affect the outcome so much that incorrect evaluation of these properties would severely compromise the results.

5.4.2 Mutations

The GA synthesizes the design process by introducing architectural design patterns and styles. Naturally, the selection of available patterns and styles dictate the outcome, which is, to be put plainly, the null architecture enhanced with patterns and styles. Thus, the outcome would be quite different should the pattern collection contain a different set of patterns. Two styles and five different patterns were used as mutations, and they were chosen so that as little information as possible of the implementation of operations is required, i.e., only invocation of methods could be known, as this could be deduced from the sequence diagrams. However, no object

composition or any other instantiation related information was available, and thus creational patterns were not applicable at this stage. Thus, a more interesting topic to discuss related to mutations is how the preconditions for mutations affected the outcome and did the GA favor some patterns over others?

Firstly, it should be noted that having one architecture style present in the solution did not disable the other style, i.e., both client-server style and the message dispatcher style could be used side by side. This made the algorithm more flexible, but also the produced solutions somewhat unnatural, as it is unlikely in practice to have a design which heavily uses the message dispatcher and yet suggests that the operations of one class should be accessed as services, especially if there is no significant data storage in that class. As there is a precondition related to the amount of operations in a class in order to apply the client-server style, it was used sparingly, and never quite reached the level of a "real" client-server architecture. It also appeared that the GA somewhat favored the client-server style for those classes which have no or very few calls to other classes. The server was also used to introduce indirect support for modifiability in the scenario case studies (publication [III]). These design choices can not be directly explained by the fitness function or mutation definitions, but demonstrate the power of GA: the client-server style is actually very natural to use for classes in the end of a call sequence (operations which are implemented as services do not need to call other services). The message dispatcher style, in turn, was more common for all solutions (once its use was eased by the fitness function or complementary crossover), as there were no preconditions regarding its application (except in the latest version, operations within the same null architecture class were forbidden to use the message dispatcher for communication).

Secondly, Façade and Mediator were practically never used. This was due to the structure of used sample systems and using the null architecture for the basic class structure, for which the preconditions for Façade and Mediator could not be satisfied. This most likely slowed down the evolution to some extent, as each time the mutation selection landed on a Façade or Mediator related mutation, it had no affect on the solution, which subsequently carried on the next generation as it was. Thus, if these two patterns had looser preconditions (although, in that case, they would not correspond to the original definitions) or would have been substituted by other patterns, the evolution might have performed faster and the GA converged quicker.

Thirdly, the preconditions for Template Method and Strategy had a noticeable effect on the outcome. In several studies, the GA seemed to favor one of these two patterns over the other, as the precondition was very similar for both patterns. As Template Method resulted in smaller

efficiency penalty, it was most often favored over Strategy. The Adapter pattern, in turn, was commonly applied in most studies, as it had a very easily satisfiable precondition. The preconditions also had the effect that solutions were very similar; when there were only a few practical places to apply certain patterns, the GA seemed to favor certain operations. This, of course, could be dues to the estimated properties, especially in the case of variability.

5.4.3 Fitness function

The definition of the fitness function clearly affected the outcome, as the architectures were only evaluated based on the attributes considered in the fitness function. However, as choosing the given quality attributes has been discussed already in Chapter 3, and as there is little point in discussing what the solutions would have been like had the fitness function evaluated, say, reliability, I will concentrate on other aspects of the fitness function.

The accurate calibration of the weights for the single weighted fitness function had the most drastic affect on the results. In most of the studies (publications [I-V], [VII] and [VIII]), the fitness function enabled an exponential reward for positive modifiability if the message dispatcher was extensively used. In order to ensure that the values for all subfunctions were at same range, positive modifiability was given a rather small weight in the standard experiments. Thus, the message dispatcher was very rarely seen if modifiability was not valued over other quality attributes, as the penalty for using the dispatcher was quite big (as a result of negative efficiency). Only after the application of complementary crossover (Section 4.2) could the message dispatcher be seen in the architectures.

Also, as stated before, the architectures are most efficient at null architecture stage. Negative efficiency penalizes connections between classes and using the message dispatcher and server, which are all minimized in the null architecture. At null architecture stage the message dispatcher is not present, the client-server style is not applied, and there are no “extra” connections on the account of having pattern related classes. Thus, after the evolution begins, the efficiency value will begin to decrease, and the modifiability value increase – if the subfunctions are well-defined and equally weighted. Careful assessment of the fitness function was needed to achieve this balance. For example, once the message dispatcher related reward was altered (and subsequently, the normalized weight for the modifiability sub-fitness was re-evaluated), the fitness curve began to behave oddly and was not able to reach good values at all. In order to

correct this, the pattern-related reward in the modifiability sub-fitness was introduced to the last version of the fitness function.

Finally, a significant question related to the fitness function is whether it portrays the "true" quality of a given system. The fitness function appears to be a good indicator, but single values should not be compared. A better choice is to use the multi-objective approach, and consider the architecture from several viewpoints. However, if the single weighted fitness function is used, comparison can be made using ranks. The design which has the highest fitness values can quite reliably be considered significantly better than the design at the tenth place in the population sorted by fitness values. The Pareto front related evaluation in publication [IX] also supports this.

5.4.4 Case systems

Two different sample systems were used for case studies. Analysis was made between and within cases. As stated, there were very few differences between the solutions for different cases although the sample systems were (intuitively) quite different. It was rather obvious that the GA only considered the structure of the system from the viewpoint of how easy it was to apply different patterns and increase the fitness value. The GA was, however, not able to deduce tacit information of the purpose of systems, which was portrayed in the solutions.

The solutions for ehome rarely contained the message dispatcher because the classes were quite large, and most of the dependencies between operations were within-class calls, and could not use the message dispatcher. The structure for ehome also enabled the use of several Strategy and Template Method patterns; some solutions had as many as six low-level patterns involved with one class, e.g., the CoffeeMachine.

For robo, where the structure was different (many small classes and dependencies between classes), the effect was naturally the opposite. The message dispatcher was often found, and there were usually just one or two patterns per class. One remark concerning the use of client-server: in Subsection 5.3.2 it was stated that the GA favored this style for classes at the end of a call sequence, which mostly corresponds to results for ehome. However, for robo, the client-server style was firstly much less used than with ehome, and secondly, it was also commonly applied to classes in "mid-chain" (in addition to those at the end of call sequence). This is due to the fact that for robo, the classes at the end of a call sequence usually contained very few operations, and thus this mutation was not applicable, because of the preconditions.

The message dispatcher style is a good example of how the GA is not able to detect the particularities of the system under design. The message dispatcher would actually be quite a desired architecture solution for ehome, but not so much for robo. However, as the GA is only given the crude structure of these systems, and no information about what is suitable for what kind of system, in the used cases it applied the message dispatcher style in a completely opposite way than a human designer would.

A significant point regarding the case studies is that, in practice, the quality requirements for the used sample systems would most likely not be completely equally weighted, but, as discussed in Section 1.4, for robo efficiency would be important, and ehome would probably be mostly optimized towards modifiability. Thus, the set up used in the different experiments was not entirely natural, as in order to achieve comparable results, all quality attributes were weighted equally. If the synthesis was performed with the aim of producing as good a real-life system as possible, quality attributes would be weighted more realistically, or the multi-objective approach would be used. For example, obvious choices such as how to use the message dispatcher, could be relayed to the GA in the form of weights (e.g., for ehome, overweight modifiability, for robo, overweight efficiency) or by adjusting the mutation probabilities (e.g., very low dispatcher probability for robo). In the presented case studies, however, this has not been done, as the aim was to achieve unbiased results and see how "intelligent" the synthesizer is on its own.

Naturally, using more cases would have significantly increased the level of confidence in evaluation and further confirmed the results. However, in order to actually get new results the cases should be carefully selected. Firstly, the structure should enable the use of Mediator and Façade, in order to study how the GA performs when such middle-level patterns are actually applicable. Secondly, the case system should be significantly larger in size than ehome or robo. The large size would most probably also better enable using the mid-level patterns. Finally, it would be good if the quality attributes would actually be balanced (unlike with ehome and robo), or if there would be no clear intuition as to what is the best solution, and thus the GA solutions could truly be taken as suggestions as they are.

Bringing new case(s) would provide new light to this research. Mainly, is the GA able to sophisticatedly introduce mid-level patterns, or are they used simply because it is possible? A larger system would show how effective the evolution process actually is – does the GA require a longer time (more generations) in order to produce satisfying results when intuitively there is more work to be done?

However, while more cases would bring valuable information of the synthesis, the performed case studies are quite sufficient in showing the applicability of genetic algorithms for pattern-based software architecture synthesis. The two cases are medium-sized systems which differ in structure and purpose, and thus provide information on how the algorithm performs in two very different situations, and means to deduce how the GA will most likely perform for systems where the structure is somewhere in between those used here.

5.4.5 Limits and potential

There are, naturally, many limits to the synthesis process at this stage. The synthesis is dependent on the selection of patterns, and thus completely realistic architectures are not available, as this small selection of design patterns is rarely sufficient. The selection of quality attributes used for evaluation is also very limited, as architectures are usually evaluated from many different viewpoints. Most importantly, the GA does not have the ability to identify concepts within the systems under design: it only recognizes the information which is explicitly given to it.

Thus, the results of the experimental study discussed in Section 5.2 are probably a result of these limitations: should the GA be provided with a wider selection of tools (patterns) and more information of the system, the results would most likely far exceed those of the third year student. While the students were given instructions to concentrate only on certain quality aspects and a limited amount of information regarding the nature of the system, these recommendations naturally did not inhibit the students from also considering the design task from other viewpoints as well and using intuitive knowledge not available for the GA.

The GA does, however, have a lot of potential. It is already significantly faster than the average student. The GA is also unbiased – while not being able to use intuitive knowledge is a weakness, it is partly a strength as well, as the GA does not care about how things have been usually done before, it only cares about increasing the fitness values. The multi-objective GA is also able to produce several competing solutions for the same quality requirements at once, with different emphasis on the requirements. No human would instantly know how to adjust the design if the emphasis on quality requirements would suddenly change.

Thus, theoretically, the actual limits to how far the synthesis can be taken are still quite far away. The pattern collection could be significantly increased, if the patterns would be stored into a kind of database, and the mutations would not be hardcoded, as they are in the current implementation. Further improving the scenario-based evaluation would make it easier to add other quality attributes in order to produce a more

detailed evaluation – while transforming some quality requirements to metrics might be nearly impossible, transforming them into simple scenarios is always possible and should not even be too complex, as this is done all the time in real-life evaluation. As for providing the GA more information about the system: anything can be encoded using the current supergene format. For example, the supergene could be extended with a field "system information", and constant variables could be defined to introduce different types of information. This could be introduced to the GA in the form of a dummy gene only containing this information (as with the message dispatcher). The GA could then consider this extra information if it affects the application of mutations.

At its current stage the synthesizer could already be used as a way to produce a kind of "jump-start solution". An appealing idea relying on this kind of initially synthesized solution is incremental design, where the human and synthesizer take turns in designing the architecture. The initial solution would be a synthesized one, and the human could freeze the design solutions (patterns and styles) which are particularly good. The human architect could then guide the synthesizer by adding some new design solutions, and freezing them as well. This improved solution could then be given to the synthesizer, which would further enhance the architecture. Repeating these steps would ultimately produce an architecture which satisfies all quality requirements. This kind of incremental design is already supported by tools such as ArchE [McGregor et al., 2007], which, however, uses deterministic methods for design. The GA, in turn, should be able to produce much more imaginative solutions, as it is not required to follow a direct path in the design, but can search for better options from a much wider spectrum. Interestingly, the ArchE tool uses very similar input to what is given for the GA synthesizer. Both use the class structure of a system as a base, and use encoded quality requirements as means to guide the development.

To summarize, there are currently many factors that affect the quality of synthesized solutions. However, many of these factors can be dealt with, and bringing the synthesis to a higher level is realistic. While the lack of intuitive knowledge is also a great strength for the GA, it is still its main weakness. Thus, it is rather unlikely to expect that software architects would accept a synthesized solution "as it is". Hence, the synthesizer may be viewed as a method for jump starting the design process, and significantly easing the work load of human software architects, who would still be required for making the final decision of what design to use.

6 Conclusions

This chapter concludes the thesis. As the main results have been discussed at length in the previous chapter, here the focus is on the actual research questions presented at the beginning of this thesis. The questions are revisited and answered in the following.

1. *What information of the system (and its architecture) under design is required as input for the GA in order to perform the synthesis?*

Results show that a class structure in the form of a null architecture accompanied with a call sequence is quite sufficient for this kind of pattern-based approach. The null architecture can be deterministically elicited from use cases and sequence diagrams. Additional properties for different operations, such as frequency of use and sensitiveness to variation are extremely beneficial, and help improve the quality of solutions, but similar information could be encoded, e.g., as scenarios. As discussed, the class structure provided by the null architecture should be fixed. The GA is not able to handle both pattern selection and class structure optimization. Class responsibility assignment, as shown, is a completely different, highly complex design problem.

If the design should be further optimized, some information of the type of the system under design would be beneficial to include. The case studies were made on an embedded system and framework system, but the GA was unable to deduce this type of tacit information, and thus some design solutions were quite contradictory to human intuition. Thus the ultimate answer from the viewpoint of this thesis would be: a class structure and call sequence between operations are satisfactory and the only things actually needed, however, including operation properties would be very

useful, and further including higher-level information of system purpose could be beneficial.

2. *How can the architecture be numerically evaluated in the fitness function? The GA is not a human with hidden knowledge, and it needs a well-defined formula to calculate the quality of a proposed solution. Are current software metrics sufficient for this, and what kinds of methods are needed to achieve accurate evaluation?*

In order to achieve accurate evaluation, several detailed issues need to be considered. Firstly, as much information as possible should be elicited from the system, i.e., if possible, the different properties of operations as discussed with Question 1 should be used. Secondly, as much information as possible regarding how different design solutions affect quality attributes should be elicited. This is apparent in the current definition of the fitness function, as given in Subsection 3.1.4, as there are several arguments regarding the message dispatcher and server use. These detailed design decisions are not considered by general object-oriented metric sets, but are crucial in accurate evaluation of synthesized solutions. Finally, the level of detail can be increased by adding scenario evaluations, as discussed in Section 4.3. Once all these things are considered, metrics can be tuned so that accurate evalution of the architecture is achieved, as demonstrated by the different case studies.

3. *Is a traditional and simple GA enough, or should some more complex operations be studied, in order to comply with the problem of software architecture design?*

A traditional GA is not enough. Firstly, the encoding mechanism itself differs from those used with simpler problems: the encoding uses several fields for each gene instead of just one, and some fields contain instances which in turn hold more data particles than just one. Secondly, a simple random crossover is not the best choice, as both asexual reproduction (in terms of speed) and the complementary crossover (in terms of quality) performed significantly better than a random crossover. Finally, the GA should be multi-objective, and not rely on a single weighted fitness function. While multi-objective approaches are becoming more common, there is still a big difference to the traditional, “simple” GA.

4. *How far can the automated design be taken? What is the level of quality that can be achieved with automation?*

The most interesting question – what is the actual level of quality that can be achieved, and how far can the automation be taken? The simple, one-line answer would be, the level of a third year software engineering student, as discovered in the experiment presented in Section 5.3. However, this is not the whole truth, and this question should be discussed at some length.

Firstly, GA-based software architecture design at this level is itself a novel concept. Thus, the fact that automation would be possible to some extent at all, is already remarkable. It is, in fact, possible to define the fitness function and operations in such a way that given a class diagram as a starting point, the GA will produce sensible solutions.

Secondly, the GA showed quite surprising problem solving capabilities in particularly when used with the scenario fitness extension. When the GA was aware that some operations were especially crucial for the design, it attempted to "help" these operations in any possible way – even indirectly.

Thirdly, while some may consider it quite a troubling thought that the GA is only at the level of a third year engineering student, it is not actually a bad achievement. The GA only relies on a fixed fitness function, has no higher-level knowledge of the systems under design, and as a result, no intuition of generally accepted design choices either. The students, however, have a lot of tacit information not available for the GA – and as a reminder, several students today are employed by software companies at an early stage of their studies already. Compared to this, my GA is performing quite well considering that it is practically only a newborn in the field of software architecture design.

Finally, to what extent could the design process be automated, if the GA was further improved? In my view, the design process can be automated so much that architects are only required to press the button for a synthesizer and make a decision on what solution of the range of options provided by the GA is the ultimately best one. This, of course, still requires much improvement. However, if with a limited set of patterns, a fairly minimal amount of information as input, and a crude fitness function the GA is able to reach the level of a nearly ready B.Sc., then I see no reason why it would not be able to reach the level of a software engineering expert. This kind of optimistic view does, of course, require that the collection of patterns is significantly increased, some tacit information of the system is made available, and that the fitness function is further developed to meet the requirements of the new design choices.

It can clearly be seen that the better results we want to achieve with pure synthesis, the more information we need to provide the GA with. However, eliciting any new information of the system under design requires work effort from the architect, as more detailed evaluation of the underlying requirements must be conducted. The more work is needed to use the synthesizer, the less it fills its purpose as an aid to the architect. Thus, either the GA should be tuned so that better results can be achieved without too much additional effort from the architect, or the synthesized solution is considered more as a jump-start to the design process, rather than a finalized solution.

To conclude, in a utopistic dream it is possible to refine the synthesizer so that the architect need nothing more than push a button and a beautiful design would be produced within minutes. However, as this ideal world is still rather far away (although, in my view, not completely unreachable), at this point it may be more realistic to picture the synthesizer working side by side with the architect, providing means for incremental design. Thus, the GA-based approach should even in its current state be able to produce quality designs, if it is just nudged to the proper direction at certain intervals. With further development to the synthesizer, the need for these nudges should decrease, until finally the GA will be able to produce quality designs on its own.

7 Introduction to Publications

This thesis contains an introductory part and nine publications. The publications are briefly summarized here, as well as the author's contribution to each publication.

[I] O. Räihä, K. Koskimies and E. Mäkinen, Genetic Synthesis of Software Architecture, In: *Proc. of the 7th International Conference on Simulated Evolution and Learning (SEAL'08)*, Melbourne, Australia. December 2008, Springer LNCS **5361**, 565-574.

In publication [I] the foundations for the research are defined. Genetic synthesis of software architecture is presented so that the algorithm only takes as input the call sequence of operations and uses message dispatcher architecture style along with two patterns for mutations, as well as decomposition and interface introduction. The author of this thesis was the main author in this publication, and responsible for implementation and experiments. Professors Koskimies and Mäkinen acted as supervisors and were involved with designing the experiments.

[II] O. Räihä, K. Koskimies, E. Mäkinen and T. Systä, Pattern-Based Model Refinements in MDA, *Nordic Journal of Computing*, **14** (4), 2008, 338-355.

In publication [II] domain knowledge is first used to aid the GA in building the architecture. The approach is now extended so that it is applicable for MDA, and the amount of mutations is increased with three new patterns and the client-server architecture style to enable more varied solutions. The author of this thesis was the main author in this publication, and responsible for implementation and

experiments. Professors Koskimies and Mäkinen acted as supervisors and were involved with designing the experiments, while Professor Systä acted as supervisor and the MDA expert.

[III] O. Räihä, K. Koskimies and E. Mäkinen, Scenario-Based Genetic Synthesis of Software Architecture, In: *Proc. of the 4th International Conference on Software Engineering Advances (ICSEA'09)*, Porto, Portugal. September 2009, IEEE CS Press, 437-445.

In publication [III] scenarios are used for a more detailed evaluation of architecture. A new pattern is added to the pattern collection used for mutations. The author of this thesis was the main author in this publication, and responsible for implementation and experiments. Professors Koskimies and Mäkinen acted as supervisors and were involved with designing the experiments.

[IV] O. Räihä, K. Koskimies and E. Mäkinen, Empirical Study on the Effect of Crossover in Genetic Software Architecture Synthesis, In: *Proc. of the World Congress in Nature and Biologically Inspired Computing (NaBiC'10)*, Coimbatore, India. December 2009, IEEE Press, 619-625.

In publication [IV] a new kind of reproduction method, "asexual reproduction" was experimented, as a random crossover seemed insufficient for the synthesis task. The author of this thesis was the main author in this publication, and responsible for implementation and experiments. Professors Koskimies and Mäkinen acted as supervisors and were involved with designing the experiments.

[V] Hadaytullah, S. Vathsavayi, O. Räihä and K. Koskimies, Tool Support for Software Architecture Design with Genetic Algorithms, In: *Proc. of the 5th International Conference on Software Engineering Advances (ICSEA'10)*, Nice, France. August 2010, IEEE CS Press, 359-366.

Publication [V] presents tool support for the synthesizer. The author of this thesis was responsible for the underlying genetic algorithm which performs the synthesis and interfacing it with the tool, while Mr. Hadaytullah (M.Sc.) and Mr. Vathsavayi (M.Sc., tech) were mainly responsible for experiments and implementing the user interface. Professor Koskimies acted as supervisor and was involved with designing the experiments.

[VI] O. Räihä, A Survey on Search-Based Software Design, *Computer Science Review*, **4** (4), 2010, 203-249.

Publication [VI] provides a comprehensive survey of studies in search-based software design. The author of this thesis was the sole author of this publication.

[VII] O. Räihä, K. Koskimies and E. Mäkinen, Complementary Crossover for Genetic Software Architecture Synthesis, In: *Proc. of the International Conference on Intelligent Systems Design and Application (ISDA'10)*, Cairo, Egypt. December 2010, IEEE Press, 260-265.

Publication [VII] presents yet another kind of reproduction method, complementary crossover, which is implemented in two ways. The author of this thesis was the main author in this publication, and responsible for implementation and experiments. Professors Koskimies and Mäkinen acted as supervisors and were involved with designing the experiments.

[VIII] O. Räihä, Hadaytullah, K. Koskimies and E. Mäkinen, Synthesizing Architecture from Requirements: A Genetic Approach, In: P. Avgeriou, J. Grundy, J. G. Hall, P. Lago and I. Mistrik (eds), *Relating Software Requirements and Architectures*, Springer, to appear.

Publication [VIII] presents a comprehensive view of the genetic synthesis process. The process is depicted in more detail, with emphasis on collecting requirements to give as input for the GA. This publication also discusses an empirical study where synthesized solutions are compared to student solutions. The author of this thesis was the main author in this publication, and responsible for implementation and experiments. Mr. Hadaytullah (M.Sc.) was mostly responsible for matters related to requirement gathering, and Professors Koskimies and Mäkinen acted as supervisors and were involved with designing the experiments.

[IX] O. Räihä, K. Koskimies and E. Mäkinen, Generating Software Architecture Spectrum with Multi-Objective Genetic Algorithms, In *Proc. of the Third World Congress on Nature and Biologically Inspired Computing (NaBIC'11)*, Salamanca, Spain, October 2011, IEEE Press, to appear.

Publication [IX] presents a multi-objective approach for genetic software architecture synthesis. Pareto optimality is applied, and a palette of solutions is achieved. The Pareto fronts are also evaluated against scenarios to compare how the fitness values correspond to human evaluation. The author of this thesis was the main author in this publication, and responsible for implementation and

experiments. Professors Koskimies and Mäkinen acted as supervisors and were involved with designing the experiments.

8 References

[Aleti et al., 2009] A. Aleti, S. Björnander, L. Grunske and I. Meedeniya, ArcheOpterix: an extendable tool for architecture optimization of AADL models. In: *Proceedings of the ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2009, 61–71.

[Amoui et al., 2006] M. Amoui, S. Mirarab, S. Ansari and C. Lucas, A genetic algorithm approach to design evolution using design pattern transformation. *International Journal of Information Technology and Intelligent Computing*, **1** (1, 2), 2006, 235–245.

[Bansiya and Davis, 2002] J. Bansiya and C.G. Davis, A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, **28** (1), 2002, 4-17.

[Bass et al., 1998] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 1998.

[Booch, 1991] G. Booch, *Object-Oriented Design – with Applications*. The Benjamin/Cummings Publishing Company, 1991.

[Bosch, 2000] J. Bosch, *Design & Use of Software Architectures – Adopting and Evolving a Product-Line Approach.* Addison-Wesley, 2000.

[Bowman et al., 2010] M. Bowman, L.C. Briand and Y. Labiche, Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Transactions on*

Software Engineering, **36** (6), 2010, 817–837.

[Briand et al., 2000] L.Briand, J. Wüst, J. Daly and V.Porter, Exploring the relationships between design measures and software quality in object oriented systems. *Journal of Systems and Software*, **51**, 2000, 245–273.

[Chidamber and Kemerer, 1994] S.R. Chidamber and C.F. Kemerer, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, **20** (6), 1994, 476–492.

[Clarke et al., 2003] J. Clarke, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper and M. Shepperd, Reformulating software engineering as a search problem. *IEE Proceedings - Software*, **150** (3), 2003, 161–175.

[Clements et al., 2002] P. Clements, R. Kazman and M. Klein, *Evaluating Software Architectures*. Addison-Wesley, 2002.

[Deb, 1999] K. Deb, Evolutionary algorithms for multicriterion optimization in engineering desing. In: *Proceedings of EUROGEN'99*, 1999, 135-161.

[Di Penta et al., 2005] M. Di Penta, M. Neteler, G. Antoniol and E. Merlo, A language-independent software renovation framework. *Journal of Systems and Software*, **77**, 2005, 225–240.

[Doval et al., 1999] D. Doval, S. Mancoridis and B.S. Mitchell, Automatic clustering of software systems using a genetic algorithm. In: *Proceedings of the Software Technology and Engineering Practice*, 1999, 73–82.

[Du Bois and Mens, 2003] B. Du Bois and T. Mens, Describing the impact of refactoring on internal program quality. In: *Proceedings of the International Workshop on Evolution of Large-Scale Industrial Software Applications* 2003, 37–48.

[Eclipse, 2011] http://www.eclipse.org, checked 18.2.2011.

[Evesti, 2007] A. Evesti, Quality-oriented Software Architecture Development. M.Sc. (tech) thesis, Helsinki University of Technology. VTT publication 636, 2007.
http://www.vtt.fi/inf/pdf/publications/2007/P636.pdf, checked

15.3.2011.

[Frankel, 2003] D.S. Frankel, *Model Driven Architecture – Applying MDA to Enterprise Computing*. Wiley, 2003.

[Gamma et al., 1995] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GraphViz, 2011] http://www.graphviz.org, checked 18.2.2011.

[Harman et al., 2009] M. Harman, S.A. Mansouri and Y. Zhang, Search based software engineering: a comprehensive review of trends, techniques and applications. Technical report TR-09-03, King's College, London, United Kingdom, 2009.

[Harman and Tratt, 2007] M. Harman and L. Tratt, Pareto optimal search based refactoring at the design level. In: *GECCO 2007: Proceedings of the Genetic and Evolutionary Computation Conference,* 2007, 1106–1113.

[Holland, 1975] J.H. Holland, *Adaption in Natural and Artificial Systems*. MIT Press, Ann Arbor, 1975.

[ISO, 2007] *Systems and software engineering — Architecture description*. ISO/IEC 42010: 2007.

[Järvinen, 1999] P. Järvinen, *On Research Methods*. Opinpaja Oy, 1999.

[Jensen and Cheng, 2010] A.C. Jensen and B.H.C. Cheng, On the use of genetic programming for automated refactoring and the introduction of design patterns. In: *Proceedings of the 2010 Genetic and Evolutionary Computation Conference (GECCO'10)*, 2010, 1341–1348.

[JFreeChart, 2011] http://www.jfree.org/jfreechart, checked 18.2.2011.

[Kleppe et al., 2003] A. Kleppe, J. Warmer and W. Bast, *MDA Explained – The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[van Laarhoven and Aarts, 1987] P.J.M. van Laarhoven and E. Aarts, *Simulated Annealing: Theory and Applications*. Klüwer, 1987.

[Losavio et al., 2004] F. Losavio, L. Chirinos, A. Matteo, N. Lévy and A. Ramdane-Cherif, ISO quality standards for measuring architectures. *The Journal of Systems and Software* **72**, 2004, 209–223.

[Martens et al., 2010] A. Martens, H. Koziolek, S. Becker and R. Reussner, Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: *Proceedings of the ACM Workshop on Software and Performance (WOSP) / SPEC International Performance Evaluation Workshop (SIPEW)*, 2010, 105–116.

[Matinlassi, 2006] M. Matinlassi. Quality-driven Software Architecture Model Transformation. Towards automation. Ph.D. thesis, University of Oulu. VTT, 2006. http://www.vtt.fi/inf/pdf/publications/2006/P608.pdf checked 15.3.2011.

[Mens and Demeyer, 2001] T. Mens and S. Demeyer, Future trends in evolution metrics. In: *Proc. Int. Workshop on Principles of Software Evolution*, 2001, 83–86.

[Michalewicz, 1992] Z. Michalewicz, *Genetic Algorithms* + *Data Structures* = *Evolutionary Programs.* Springer-Verlag, 1992.

[Miller and Goldberg, 1995] B. L. Miller and D. E. Goldberg, Genetic algorithms, tournament selection, and the varying effect of noise. *Complex Systems*, **9** (3) , 1995, 193–212.

[Mitchell, 1996] M. Mitchell, *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[O' Keeffe and Ó Cinnéide, 2006] M. O'Keeffe and M. Ó Cinnéide, Search-based software maintenance. In: *Proceedings of CSMR'06*, 2006, 249–260.

[O' Keeffe and Ó Cinnéide, 2008] M. O'Keeffe and M. Ó Cinnéide, Search-based refactoring for software maintenance, *Journal of Systems and Software*, **81** (4), 2008, 502–516.

[Olague et al., 2007] H. Olague, L.H. Etzkorn, S. Gholston and S. Quattlebaum, Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed

using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering*, **33** (6), 2007, 402–419.

[Praditwong et al., 2011] K. Praditwong, M. Harman and X. Yao, Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, to appear.

[Quaum and Heckel, 2009] F. Quaum and R. Heckel, Local search-based refactoring as graph transformations. In: *Proceedings of the 1st Symposium on Search-Based Software Engineering*, 2009, 43–46.

[Räihä et al., 2010] O. Räihä, E. Mäkinen and T. Poranen, Simulated Annealing for Aiding Genetic Algorithm in Software Architecture Synthesis. Technical Report D-2010-19, Department of Computer Sciences, University of Tampere. http://www.cs.uta.fi/reports/dsarja/D-2010-19.pdf

[Reeves, 1995] C. R. Reeves, *Modern Heuristic Techniques for Combinatorial Problems.* McGraw-Hill, 1995.

[Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Sahraoui et al., 2000] H.A. Sahraoui, R. Godin and T. Miceli, Can metrics help bridging the gap between the improvement of OO design quality and its automation? In: *Proc. of the International Conference on Software Maintenance (ICSM '00)*, 154–162.

[Seng et al., 2005] O. Seng, M. Bauyer, M. Biehl and G. Pache, Search-based improvement of subsystem decomposition. In: *GECCO 2005: Proceedings of the Genetic and Evolutionary Computation Conference*, 2005, 1045–1051.

[Seng et al., 2006] O. Seng, J. Stammel and D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems. In: *GECCO 2006: Proceedings of the Genetic and Evolutionary Computation Conference*, 2006, 1909–1916.

[Shaw and Garlan, 1996] M. Shaw and D. Garlan, *Software Architecture – Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[Simons et al., 2010] C.L. Simons, I.C. Parmee and R. Gwynllyw, Interactive, evolutionary search in upstream object-oriented class

design. *IEEE Transactions on Software Engineering*, **36** (6), 2010, 798–816.

[UMLGraph, 2011] http://www.umlgraph.org, checked 18.2.2011.

[UML2Tools, 2011] http://www.eclipse.org/modeling/mdt, checked 18.2.2011.

[Weiss, 1998] M. A. Weiss, *Data Structures and Problem Solving Using Java*. Addison-Wesley, 1998.

[Wojcik et al., 2006] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord and B. Wood, Attribute-Driven Design (ADD), Version 2.0. Technical report CMU/SEI-2006-TR-023, 2006. http://www.sei.cmu.edu/reports/06tr023.pdf checked 15.3.2011.

[Yin, 2003] R. K. Yin, *Case Study Research – Design and Methods*. SAGE Publications, 2003.

[Zeh and Zeh, 1996] J.A. Zeh and D.W. Zeh, The evolution of polyandry. 1. Intragenomic conflict and genetic incompatibility. *Proc. Roy. Soc. Lond*. **B 263**, 1996, 1711-1717.

Paper I

Outi Räihä, Kai Koskimies and Erkki Mäkinen, Genetic Synthesis of Software Architecture, In: *Proc. of the 7th International Conference on Simulated Evolution and Learning (SEAL'08)*, Melbourne, Australia. December 2008, Springer LNCS **5361**, 565-574.

© Springer, 2008. Reprinted, with kind permission from Springer Science+Business

Genetic Synthesis of Software Architecture

Outi Räihä*, Kai Koskimies** and Erkki Mäkinen*

*University of Tampere, Finland
**Tampere University of Technology, Finland
outi.raiha@uta.fi

Abstract. Design of software architecture is intellectually one of the most demanding tasks in software engineering. This paper proposes an approach to automatically synthesize software architecture using genetic algorithms. The technique applies architectural patterns for mutations and quality metrics for evaluation, producing a proposal for a software architecture on the basis of functional requirements given as a graph of functional responsibilities. Two quality attributes, modifiability and efficiency, are considered. The behavior of the genetic synthesis process is analyzed with respect to quality improvement speed, the effect of dynamic mutation, and the effect of quality attribute prioritization. Our tests show that it is possible to genetically synthesize architectures that achieve a high fitness value early on.

Keywords: architecture synthesis, genetic algorithm, search-based software engineering, software design

1 Introduction

A persistent dream of software engineering is to be able to automatically produce software systems based on their requirements. While the synthesis of executable programs is in general beyond the limits of current technology, the automated derivation of architectural designs of software systems is conceivable. This is due to the fact that architectural design largely means the application of known standard solutions in a combination that optimizes the quality properties (like modifiability and efficiency) of the software system. These standard solutions are well documented as architectural styles [1] and design patterns [2]. In addition, architectural design is guided by general principles like decomposition and usage of interfaces. Here we call all these solutions jointly (*architectural*) *patterns.*

In this paper we study the application of genetic algorithms [3] to software architecture synthesis. Architectural patterns provide a natural interpretation for genetic operations: a mutation can be realized as either the application or removal of an architectural pattern, and crossover operation can be realized by merging two architectures without breaking existing pattern instances. Fitness function can be expressed in terms of quality metrics available in the literature. Evaluating an architecture is a multi-criteria problem, but in the present implementation we have decided to add the

objective functions together using the weighted sum approach. This is done for making the implementation more efficient. Our focus is on developing the required techniques for genetic architecture synthesis, and on the investigation of the overall behavior of the genetic architecture synthesis process. The proposed architecture is produced as a UML class diagram with (possibly stereotyped) classes, interfaces and their mutual dependencies.

The main contributions of this work are a setup for genetic pattern-based software architecture synthesis starting from abstract requirements, and experimental analysis of the behavior of the genetic synthesis process, especially with regard to the development of fitness values. The former includes an approach to represent functional requirements as a responsibility graph, techniques for representing architectural information as genes, for computing quality based fitness, and for architectural crossover and dynamic pattern-based mutation, and a demonstration of the genetic synthesis using an exemplary set of architectural patterns. The latter includes an analysis of the quality improvement speed, the effect of dynamic mutations, and the effect of prioritized quality attributes.

2 Related Work

Although our work differs significantly from what has been previously done in the field of search-based software engineering, genetic algorithms have been used widely for software clustering and refactoring.

The goal of software clustering or module clustering is to find the best grouping of components to subsystems in an existing software system. The problem is to partition the graph so that the clusters represent meaningful subsystems.

The genetic algorithm presented by Clarke et al. [4] for the clustering problem is quite straightforward: the main challenge is to find a suitable encoding, after which traditional mutation and crossover operators are used.

Harman et al. [5] approach the clustering problem from a re-engineering point of view: after maintaining a system its modularization might not be as good as it was when it was taken to use.

Seng et al. [6] represent the system as a graph, where the nodes are either subsystems or classes, and edges represent containment relations (between subsystems or a subsystem and a class) or dependencies (between classes). In this application each gene represents a subsystem, and each subsystem is an element of the power set of classes.

Systems refactoring is a more delicate problem than module clustering. When refactoring a system, there is the risk of changing the behavior of a system by, e.g., moving methods from a subclass to an upper class [7]. Hence, the refactoring operations should always be designed so that no illegal solutions will be generated.

O'Keeffe and Ó Cinneide [8] define the refactoring problem as a combinatorial optimization problem: how to optimize the weighting of different software metrics in order to achieve refactorings that truly improve the system's quality. Seng et al. [7] have a similar approach, as they attempt to improve the class structure of a system by moving attributes and methods and creating and collapsing classes. O'Keeffe and Ó

Cinneide [9] have continued their research with the use of the representation and mutation and crossover operators introduced by Seng et al. [7].

Harman and Tratt [10] introduce a more user-centered method of applying refactoring. They offer the user the option to choose from several solutions produced by the search algorithm.

Amoui et al. [11] have applied genetic algorithms for finding the optimal sequence of design pattern transformations to increase the reusability of a software system. Our approach is similar to Amoui et al.'s work in that we use high-level structural units, patterns, as the basis of mutations in a genetic process. We have also applied their supergene idea, to be discussed in Section 3, as a starting point for representing the architecture. However, there are several differences.

First, we consider not only reusability (or modifiability) as the quality criteria, but in principle we are interested in the overall quality of the architecture. In this paper we focus on two quality attributes, efficiency and modifiability.

Second, we aim at the synthesis of the architecture starting from requirement-level information, rather than at improving an existing architecture. Third, we do not restrict to design patterns, but consider more generally various kinds of architectural solutions at different levels.

Our viewpoint is different from that of system clustering and refactoring. System clustering considers software architecture only from the decomposition perspective, and software refactoring aims at structural fine-tuning of software architecture, whereas our approach strives for automating the entire architecture design process.

3 Technique for Genetic Architecture Synthesis

3.1 Representing functional requirements

A major problem in automated software architecture synthesis is the representation of functional requirements. We have adopted here an approach where functional requirements are represented as a responsibility dependency graph, each node representing a responsibility, and each directed edge representing a dependency between the two responsibilities. Here a responsibility denotes an elementary task or duty that can be identified for the system by analyzing its functional requirements (e.g. use cases). A responsibility depends on another responsibility if the former needs the latter to fulfill its task. These responsibilities remain as elements of the architecture as they are assigned to interfaces and classes, although they carry no semantics as far as the architecture synthesis is concerned. The architecture produced by the genetic synthesis reflects functional requirements only to the extent the responsibilities have been identified.

To allow the evaluation of the quality (here efficiency and modifiability) of the architecture, the responsibilities can (but do not have to) be associated with values characterizing, e.g., parameter size, time consumption, and the variability factor of the responsibility. However, in this paper, we assume that the values can be derived from the requirements; the quality of the results of this technique depends on the accuracy of these values. The given values for the attributes are relative, rather than absolute.

In this work we have used an intelligent home system as a case study [12]. Such a system provides an infrastructure and interfaces for controlling various home devices, like lights, drapes, and media devices. A fragment of the responsibility dependency graph of this system is depicted in Figure 1, where the dependencies between and names of responsibilities are shown, as well as property values for variability factor, parameter size and time consumption (in this order). For example, the drapeState responsibility is a data responsibility, marked with a thicker line. In the middle of the graph is the responsibility CalculateOptimalDrape, which has a variability of 3, as the optimal drape position can be computed differently in different types of homes. As it is a heavy operation, its parameter size and time consumption are also among the highest values of those shown here. Responsibilities with such high attribute values play an important role when constructing quality architecture. The entire responsibility set for this system contains 52 responsibilities and 90 dependencies between them.

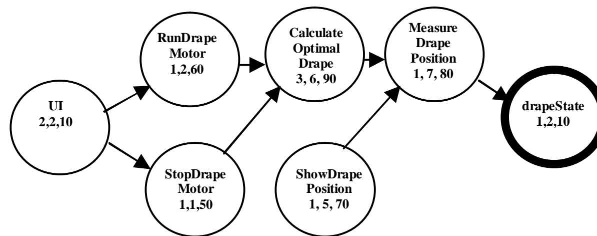


Figure 1. Fragment of a responsibility dependency graph

3.2 Architectural patterns

In the context of the present paper, an architectural pattern can be any general structural solution applied at the architectural level to improve some quality attribute of the system. Each architectural pattern gives rise to two mutation operations: introducing and removing the pattern.

In our experiments, we have used the following list of architectural patterns:

- decomposing a component
- using an interface
- Strategy design pattern [2]
- Façade design pattern [2]
- message dispatcher architectural style [1]
- communication through a dispatcher.

This collection of architectural patterns is of course very small, and intended only for experimentation purposes. We wanted to cover different levels of architectural patterns: basic practices, low-level design patterns (Strategy), medium-level design patterns (Façade), and high-level architectural styles (message dispatcher). The last architectural pattern is introduced for allowing components to join a message dispatcher introduced earlier. We expect that a real architecture synthesis tool would employ hundreds of architectural patterns.

3.3 Genetic encoding of software architecture

In order for the genetic algorithm to operate on software architecture, the architecture needs to be represented as a chromosome consisting of genes. For efficiency, in this experiment the architecture encoding is designed to suit the chosen set of architectural patterns. We have followed the supergene idea, introduced by Amoui et al. [11], where each gene has several fields to store data in. Taking this idea as a starting point, it is quite straightforward to place all information regarding one responsibility into one supergene. This also makes it easier to keep the architecture consistent, as no responsibility can be left out of the architecture at any point, and there is no risk of breaking the dependencies between responsibilities.

There are two kinds of data regarding each responsibility r_i. Firstly, there is the basic information concerning r_i given as input, containing the responsibilities $R_i = \{r_k, r_{k+1}, \ldots, r_m\}$ depending on r_i, its name n_i, type d_i, frequency f_i, parameter size p_i, execution time t_i, call cost c_i and variability v_i. Secondly, there is the information regarding the responsibility r_i's place in the architecture: the class(es) $C_i = \{C_{i1}, C_{i2}, \ldots, C_{iv}\}$ it belongs to, the interface I_i it implements, the dispatcher D_i it uses, the responsibilities $RD_i \subset R_i$ that call it through the dispatcher, and the design pattern P_i it is a part of. The dispatcher is given a separate field as opposed to other patterns for efficiency reasons. Figure 2 depicts the structure of a supergene. The actual chromosome is formed by simply collecting all supergenes [12].

R_i	n_i	d_i	f_i	p_i	t_i	c_i	v_i	C_i	I_i	D_i	RD_i	P_i

Figure 2. Supergene SG_i for responsibility r_i

Although basic operations in the architecture are relatively safe with this representation method (i.e., the responsibilities and their dependencies stay intact in the architecture), the design patterns produce challenges at the chromosome level, as careless operations can easily break patterns and make the architecture incoherent. Thus, in order to quickly check the legality of an operation with regard to patterns, a Pattern field is located in every supergene. The Pattern field has as attributes the classes, responsibilities and the interfaces involved with that particular pattern.

An initial population is first produced, where only basic structures, such as class division and interfaces for the responsibilities are randomly chosen [12]. To ensure as wide a traverse through the search space as possible, four special cases are inserted: all responsibilities being in the same class, all responsibilities being in different classes, all responsibilities having their own interface, and all responsibilities being as much grouped to same interfaces as the class division allows.

3.4 Mutation and crossover operations

All mutations are implemented as either introducing or removing an architectural pattern, i.e., decomposition, interfaces, message dispatcher and design patterns. This ensures a free traversal through the search space, as moves that may have seemed good at one time can be cancelled later on.

All mutations except for introducing a message dispatcher or a design pattern operate purely at supergene level by changing the value of the corresponding field. Introducing a new dispatcher to the system, however, is achieved by adding a "dummy" gene with only the dispatcher field containing data. Introducing design patterns, on the other hand, operate at supergene level, but affect more than one gene.

The legality of a mutation is always checked before it is administered to the selected gene. For this purpose, "architectural laws" have been defined. In our experiments, we have used three kinds of laws. Firstly, these laws ensure uniform calls between two classes: a class can communicate with another class only in a single manner (e.g. through an interface or through a message dispatcher). Secondly, the laws state some ground rules about architecture design, for example, that a responsibility can appear at most once in an interface. Thirdly, the laws regulate the order of introduction. For instance, a dispatcher must be introduced to the system before responsibilities can use it for communication.

Mutations are given a certain probability with which they are applied. The roulette wheel method [3] is used for selecting a mutation. A "null" mutation is also possible, giving a chromosome the chance to stay intact into the next generation. In addition, to study the effect of favoring more fundamental solutions in early stages, dynamic mutation probabilities have been defined for a set of patterns (dispatcher, Façade and Strategy). After 1/3 of the generations have passed, the probability of introducing a dispatcher to a system is decreased, and the probability of introducing a Façade is increased respectively. After traversing through another 1/3 of generations, the same is done with Façade and Strategy. The hypothesis is that favoring fundamental solutions (like architectural styles) in the earlier stages of evolution leads to a stronger core architecture that can be more easily refined at later stages by lower-level solutions.

In our approach, the crossover operation is also seen as a type of mutation, and thus, it is also included in the "roulette wheel". The crossover is implemented as a traditional one-point crossover with a corrective function. In the case of overlapping patterns, the left side of the offspring is always considered to be the valid one, and the right side of the crossover point is corrected so that the whole architecture is valid.

The crossover probability increases linearly in regard to how high the fitness of an individual is in the population, which causes the probabilities of mutations to decrease in order to fit the larger crossover "slice" to the "wheel". Also, after crossover, the parents are kept in the population for selection. These actions favor strong individuals to be kept intact through generations.

The actual mutation and crossover points are selected randomly. However, we have taken advantage of the variability property of responsibilities with the strategy and dispatcher communication mutations. The chances of a gene being subjected to these mutations increase with respect to the variability value of the corresponding responsibility. This should favor highly variable responsibilities.

3.5 Fitness function

The fitness function is based on widely used software metrics [13], most of which are from the metrics suite introduced by Chidamber and Kemerer [14]. These metrics

have been used as a starting point for the fitness function, and have been further developed and grouped to achieve clear "sub-fitnesses" for modifiability and efficiency, both of which are measured with a positive and negative metric. The biggest modifications to the basic metrics include taking into account the effect of interfaces and the dispatcher architecture style. A complexity metric is added to penalize having many classes and interfaces as well as extremely large classes.

Dividing the fitness function into sub-functions answers the demands of the real world, as hardly any architecture can be optimized from all quality viewpoints. Thus, we can assign a bigger weight to the more desired quality aspect. When w_i is the weight for the respective sub-fitness sf_i, the fitness function $f(x)$ for chromosome x can be expressed as

$$f(x) = w_1 * sf_1 - w_2 * sf_2 + w_3 * sf_3 - w_4 * sf_4 - w_5 * sf_5.$$

Here, sf_1 measures positive modifiability, sf_2 negative modifiability, sf_3 positive efficiency, sf_4 negative efficiency and sf_5 complexity. The sub-fitness functions are defined as follows (|X| denotes the cardinality of X):

sf_1 = |interface implementors| + |calls to interfaces| + (|calls through dispatcher| ∗ ∑ (variabilities of responsibilities called through dispatcher)) – |unused responsibilities in interfaces| ∗ 10,

sf_2 = |calls between responsibilities in different classes|,

sf_3 = ∑ (|dependingResponsibilities within same class| ∗ parameterSize + ∑ |usedResponsibilities in same class| ∗ parameterSize + |dependingResponsibilities in same class| ∗ parameterSize)),

sf_4 = ∑ ClassInstabilities + |dispatcherCalls| ∗ ∑ callCosts, and

sf_5 = |classes| + |interfaces| + BigClassPenalty.

The multiplier 10 in sf_1 notes that having unused responsibilities in an interface is almost an architecture law, and should be more heavily penalized.

Selection of individuals for the next population is made with a roulette wheel selection, where the size of each "slice" is linearly in proportion to how high the corresponding individual's fitness is in the population. No individual can be selected more than once. Thus, the "slices" are adjusted after each selection to represent the differences between fitnesses of the remaining individuals.

4 Experiments

In this section we present the results from the preliminary experiments done with our approach, using the example system introduced in Section 3.1. The algorithm was implemented in Java 1.5, and one test run with a population size of 100 and 250 generations took approximately 90 seconds. All test runs were conducted with a fixed set of mutation probabilities, found after extensive testing. The calculated fitness value is the average of 10 best fitnesses in each generation. In all experiments, the actual y-value for the curve is achieved as the average value from five test runs. The average value is used after first ensuring a similar fitness curve for all test runs. Examples of the produced UML diagrams are presented by Räihä [12].

We first examined the overall development of the fitness values over a high number of generations. As can be seen in Figure 3, depicting the evolvement of fitness

values over 1000 generations, the fitness values achieve their highpoint after around 750 generations, and achieve quite high values already after 500 generations.

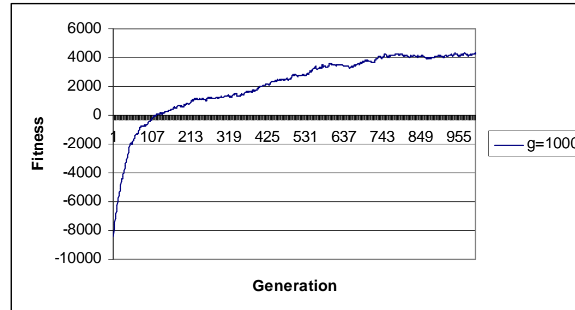


Figure 3. Fitness development

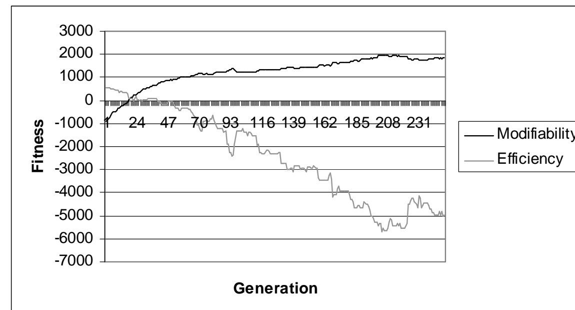


Figure 4. Heavily weighted modifiability

To analyze the effect of weighing one quality evaluator over another, we have extracted the separate sub-fitness curves for modifiability and efficiency in cases where one was weighted heavily over the other. These tests were made with a population size 100 and 250 generations. In the first test, depicted in Figure 4, the modifiability functions were weighted 10 times higher than the efficiency functions. This results in the "normal" development of the modifiability curve, while the efficiency curve plummets quite rapidly, and continues to worsen throughout the generations.

In the second test efficiency was correspondingly weighted 10 times higher than modifiability; the fitness curves are shown in Figure 5. In this case, the efficiency curve achieves very high values from the very beginning and does not develop as noticeably as the modifiability fitness in the previous case. The modifiability fitness, however, remains quite stable, achieving only low values. The explanation for the poor development of the efficiency curve lies within the special cases inserted in the initial population. As the efficiency fitness values big classes, it would assign a high fitness value for the case where all responsibilities are in the same class. From this initial case it is fairly easy to achieve individuals with very few classes and high efficiency fitness values from the very beginning.

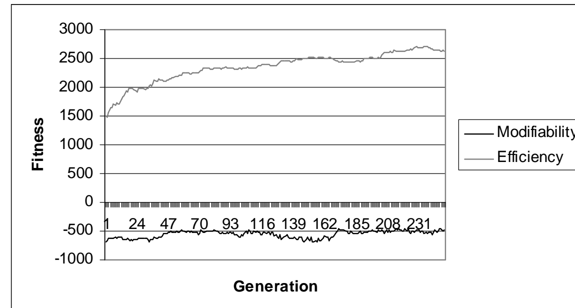


Figure 5. Heavily weighted efficiency

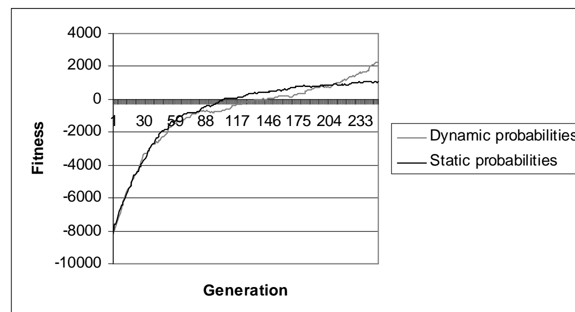


Figure 6. Pattern probability variation

Finally, the effect of dynamic mutation probabilities was analyzed by testing them against probabilities that remained the same throughout the generations; the curves for these tests are shown in Figure 6. As can be seen, with 250 generations and a population of 100, the fitness curve achieves its high point quite early when the mutations are static, but with the dynamic mutation probabilities, the fitness value continues to develop. Thus, it appears that dynamic mutation makes the basic structure of the architecture more amenable to fine-tuning in the later phases.

In this section we have shown that the quality of an architecture increases quite steadily with the selected evaluators related to modifiability, efficiency and complexity. If some quality attribute is heavily weighted in the process, it may have significant negative effect on another. Using dynamic mutation probabilities seems to offer advantages in longer generation sequences.

5 Concluding Remarks

We have presented a novel approach for genetic architecture synthesis. We have succeeded in genetically constructing an architecture from high-level responsibilities that achieves high quality values quite early in the development. The extremely rapid development of fitness values during the first 100 generations is especially notable, as it shows that a genetic algorithm can quickly find a “good” basic structure for software

architecture. As the quality of software system is largely based on its architecture, our work also brings a new level to genetic programming. In genetic programming the emphasis has been to only produce programs that perform a certain task, while the quality factors of the produced code have so far been overlooked. Our plans for future work include implementing a simulated annealing algorithm for comparison and more experiments on dynamic mutations. Moreover, we plan to implement a genuine multi-criteria approach, which would benefit in comparing the conflicting goals of modifiability and efficiency, discussed in Section 4.

References

1. Shaw, M., Garlan, D.: *Software Architecture - Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
3. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolutionary Programs. Springer-Verlag, 1992.
4. Clarke, J., Dolado, J.J., Harman, M., Hierons, R.M., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., Shepperd, M.: Reformulating software engineering as a search problem, *IEE Proceedings - Software* **150**, 3 (2003), 161-175.
5. Harman, M., Hierons R., Proctor, M.: A new representation and crossover operator for search-based optimization of software modularization. In: *Proc. GECCO 2000*, 1351–1358.
6. Seng, O., Bauyer, M., Biehl, M., Pache, G.: Search-based improvement of subsystem decomposition, In: *Proc. GECCO 2005,* 1045-1051.
7. Seng, O., Stammel, J., Burkhart, D.: Search-based determination of refactorings for improving the class structure of object-oriented systems, In: *Proc. GECCO 2006,* 1909-1916.
8. O'Keeffe, M., Ó Cinnéide, M.: Towards automated design improvements through combinatorial optimization, In: *Workshop on Directions in Software Engineering Environments – 26th ICSE,* 2004, 75-82.
9. O'Keeffe, M., Ó Cinnéide, M.: Getting the most from search-based refactoring In: *Proc. GECCO 2007,* 1114-1120.
10. Harman, M., Tratt, L.: Pareto optimal search based refactoring at the design level, In: *Proc. GECCO 2007,* 1106-1113.
11. Amoui, M., Mirarab, S., Ansari, S., Lucas, C.:A genetic algorithm approach to design evolution using design pattern transformation, *International Journal of Information Technology and Intelligent Computing* **1** (2006), 235-245.
12. Räihä, O.: Genetic Synthesis of Software Architecture, University of Tampere, Department of Computer Sciences, Licentiate thesis, 2008.
13. Sahraoui, H.A., Godin, R., Miceli, T.: Can metrics help bridging the gap between the improvement of OO design quality and its automation? In: *Proc. ICSM '00*, 2000, 154-162.
14. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* **20**, 6 (1994), 476-492.

Paper II

Outi Räihä, Kai Koskimies, Erkki Mäkinen and Tarja Systä, Pattern-Based Model Refinements in MDA, *Nordic Journal of Computing*, **14** (4), 2008, 338-355.

© Publishing Association Nordic Journal of Computing, 2008. Reprinted, with permission.

Nordic Journal of Computing 14(2008), 338–355.

PATTERN-BASED
GENETIC MODEL REFINEMENTS IN MDA

OUTI RÄIHÄ
University of Tampere
Finland
Outi.Raiha@cs.uta.fi

KAI KOSKIMIES
Tampere University of Technology
Finland
Kai.Koskimies@tut.fi

ERKKI MÄKINEN
University of Tampere
Finland
Erkki.Makinen@cs.uta.fi

TARJA SYSTÄ
Tampere University of Technology
Finland
Tarja.Systa@tut.fi

Abstract. We explore the application of genetic algorithms in model transformations that can be understood as pattern-based refinements. In MDA (Model Driven Architecture), such transformations can be exploited for deriving a PIM model from a CIM model. The approach uses design patterns as the basis of mutations and exploits various quality metrics for deriving a fitness function. A genetic representation of models and their transformations is proposed, and the genetic transformation process is studied experimentally. The results suggest that genetic algorithms provide a feasible vehicle for model transformations, leading to convergent and reasonably fast transformation process. However, more work is needed to improve the quality of the individual models produced by the technique.

ACM CCS Categories and Subject Descriptors: D.2.11 [Software Engineering]: Software Architectures; D.2.2 [Software Engineering]: Design Tools and Techniques

Key words: model transformation, genetic algorithm, model driven architecture, software design

1. Introduction

MDA (Model Driven Architecture) [13] reflects a long-standing trend in software engineering to raise the abstraction level of software system description, together with automated support for transforming high-level descriptions into lower-level ones and eventually into executable code. MDA shares many motivations for developing high-level programming languages, like the ability to use the same high-level descriptions to produce executable code for multiple platforms. In MDA, this is realized by dividing the high-level descriptions (i.e., models) into different layers, CIM (Computation-Independent Model), PIM (Platform-Independent Model) and PSM (Platform-Specific Model). Basically, the CIM level corresponds to a requirements model, while the PSM level corresponds to a detailed design model. The PIM level contains an intermediate model introducing basic platform-neutral solution structures.

Received November 2008; accepted February 2009.

However, in spite of the high expectations and years of intensive research, MDA is still far from its original vision. If the concepts of the application domain are well-understood, it is possible to map these concepts into solution mechanisms in a systematic way, and build automated transformation support based on this mapping (e.g., [8, 10, 20]). However, if the domain is not precisely fixed, the transformations require more creative design knowledge which is hard to incorporate in the transformation. This is particularly problematic at the higher abstraction levels, like in transformations from the CIM level to the PIM level.

The interpretations of the role and purpose of CIM vary significantly. It is in some cases considered as a requirements model [9, 13], capturing the requirements the user has on the system. MDA Guide [13] points out that CIM is sometimes called "a domain model" and "a vocabulary" that is familiar to the practitioners of the domain in question.

Here we adopt the view that a CIM model is a domain model of the system to be built, represented in a particular way. As customary in object-oriented software architecture design, we assume that the architectural model of the system can be derived by refining a functional model (in this case an abstract domain model) with quality-driven solutions (here patterns).

In this paper, we explore techniques to produce a solution-level (architectural) model (PIM) on the basis of a requirement-level (domain) model (CIM) automatically using genetic algorithms. In genetic algorithms [11], a "good" solution is found through simulated evolution, where individual solution generations are subject to mutation and crossover operations, competing with each other according to their "goodness", as defined by the so-called fitness function. In our case, the mutations are interpreted as specific design solutions (called patterns in this paper) applied in the transformation from the CIM level to the PIM level. The quality requirements are assumed to be encoded in the fitness function. The CIM is first used to randomly generate an initial population of solution models, this population is then submitted to evolution through hundreds of generations, and the result of the transformation is obtained as the best solution model of the last generation.

Genetic algorithms have a number of important benefits in this context. First, the level of domain-independency can be freely decided by adopting different sets of mutation patterns. If the patterns consist of, say, design patterns [6], the technique is completely domain-independent. If the patterns are intended for a particular domain, the technique is tuned for that domain, accordingly. Similarly, the level of expertise is essentially determined by the selection of patterns. In this paper we apply a small set of general-purpose design patterns as mutations. Second, the quality requirements, which are often ignored in model transformations, are naturally taken into account in this approach, since they are encoded in the fitness function. In principle this approach allows any quality requirements that can be evaluated using some computation on the models. Some quality requirements may require additional information to be associated with the models for sensible evaluation, though. Here we will concentrate on two general quality requirements, efficiency and modifiability.

Third, the genetic approach is able to produce solutions that no human designer could come up with, being free of human prejudices. Since the solution proposed

by the genetic process is guaranteed to have high quality in terms of the fitness function, it is realistic to expect that in some cases the genetic process could actually outperform a human designer.

The contributions of this paper consist of a technique to apply genetic approaches in model transformations from requirement level to solution level, and of the experimental evaluation of the technique. In particular, we are interested to study the genetic representation of models, the behavior of the genetic process, and the quality of the resulting solution models. The ultimate goal is an automated model transformation tool based on this technique. Note that the user of the tool need not understand the genetic algorithm running in the background. In the current setting, the possible quality attributes (at the moment modifiability and efficiency) are pre-defined, and only choosing their weights would be left to the user. The tool is then expected to produce a PIM that is both optimized according to the user's quality attribute preferences (and nothing else) and "legal" in terms of general design standards.

2. Related work

2.1 Model transformations

A wide range of model transformation approaches have been presented in the literature. Template-based approaches, such as XSLT [21], graph grammar based approaches like GReAT [1], VIATRA [20] and Triple Graph Grammars [17], are typically used for model-to-text but also for model-to-model transformations. A template usually binds certain aspects of the source and target metamodels and is instantiated in the actual transformation, as many times as the template matches can be found. Relational approaches, like MTF [5] and QVT [14], focus on specifying relations between model structures. VIATRA, GReAT and MTF are examples of declarative languages that specify what should be transformed, instead of how.

The approaches and tools proposed are often applicable only in relatively narrow domains. The underlying reason for this is that automatic transformation solutions with hard-wired metamodel mappings are usually aimed for. Such solutions do not take user decisions or other learning aspects into account. Instead, they always provide the same solution when applied for the same source model. In our approach, the level of domain-independence depends on the mutation patterns chosen. In this paper a sample application of our approach for a domain-independent transformation is demonstrated. In addition, our approach does not assume any hard-wired transformations.

A transformational pattern system that allows gradual and interactive model transformations is presented by the World Wide Web Consortium [21]. This approach allows variation management and human made decisions during the model transformations. The transformation specification consists of transformational patterns and assembly rules. The former describe how transformation rules are implemented, and the latter describe how the individual patterns relate and which patterns are applied to which source model elements. Flexibility and reusability is achieved through the assembly rules. This approach, as ours, aims at more flexible

transformations and increasing the level of transformation reuse. However, while it achieves flexibility by allowing user decisions during the semi-automatic transformations, our approach achieves flexibility with application of genetic algorithms.

While different PIM-to-PSM and PSM-to-code transformation approaches have been proposed, much less work has been reported in the literature on CIM-to-PIM transformations. This may be partly due to the different interpretations of the role of CIMs. Also, practical MDD-fashion transformations seem to work the better the lower the abstraction level is. The reason for this is obvious: the source models themselves are already fine-grained and well-defined, allowing the use of rather straightforward transformations. The transformations from CIMs to PIMs, on the other hand, are less intuitive and require more problem solving; the CIM essentially represents the problem domain and PIM the solution domain.

Zhang *et al.* [22] point out that since CIM-to-PIM transformations have been less practiced, converting CIM to PIM depends much on designers' personal experience or creativity. In our approach, creativeness is achieved through application of genetic algorithms. Zhang *et al.* propose a feature-oriented approach for CIM-to-PIM transformations. In that approach features are considered as key elements of CIM and components as key elements of PIM. Zhang *et al.* [22] further propose an approach to create components by clustering responsibilities that are operationalized from features.

Rodriguez *et al.* [16] consider CIM-to-PIM transformations in the domain of business process modeling. They interpret business process models to be CIMs. The trans-formation rules are defined using QVT, and their purpose is to generate UML classes and use cases that will be part of the PIM of an information system. Our approach instead is not tied to any specific domain.

2.2 Genetic algorithms

A genetic algorithm maintains a population of possible solutions. In our problem a population is a set of possible architectures which undergoes an evolutionary process imitating the natural biological evolution. In each generation better individuals have greater possibilities to survive and reproduce, while worse individuals have greater possibilities to die and to be replaced. It is believed that this process leads to a combination of the properties of the better individuals, which constitutes a good solution to the problem in question.

To operate with a genetic algorithm, one needs an encoding of possible solutions, an initial population, mutation and crossover operators, a fitness function and a selection operator for choosing the survivors for the next generation. We do not explain these concepts in detail here, because we assume that the reader is familiar with the basics of genetic algorithms, as given, e.g., by Michalewicz [11].

Genetic algorithms are widely used in problems related to software engineering [4], but most of the genetic algorithms presented in the literature solve more restricted problems than that of ours. Typical problems considered are the software clustering problem [7] and systems refactoring [12, 18]. However, system clustering considers software architecture only from the decomposition perspective, and software refactoring aims at structural fine-tuning of software architecture, whereas

our approach strives for automating the entire architecture design process.

Architectural transformations apply bigger modifications to the system than simple refactoring operations. An example of architectural transformation is the introduction of design patterns in the architecture. Amoui *et al.* [2] have earlier applied genetic algorithms for finding the optimal sequence of design pattern transformations to increase the reusability of a software system.

The present work is based on our previous system [15]. Similar to Amoui *et al.* [2] we use high-level structural units as the basis of mutations in a genetic process. We have also applied the supergene idea of Amoui *et al.* [2], to be discussed in Section 3, as a starting point for representing the architecture. However, there are several differences. First, in principle, we do not consider any specific quality criterion, but we are interested in the overall quality of the architecture. Second, we aim at the synthesis of the architecture starting from requirement-level information, rather than at improving an existing architecture. Third, we do not restrict to design patterns, but also consider more high-level architectural solutions, i.e., architectural styles [19].

We do not need any formalism (like graph grammars) to model the transformations since it is sufficient to use the supergene notations to specify the relations between the entities of the architecture.

After adopting the supergene concept, we proceed with the genetic algorithm mainly in the standard manner. However, we use dynamic mutation probabilities when introducing certain architectural solutions to the individuals of the search space. The rationale behind this policy is that favoring fundamental solutions in the earlier stages of evolution leads to a stronger core architecture that can be more easily refined at later stages by lower-level solutions.

3. Genetic model transformations

In this section we describe our approach to genetically handle the transformations that are needed to achieve a PIM from a CIM in MDA. We begin with a set of responsibilities (requirements) that can be given some relative values regarding modifiability and efficiency. Using the information given on the dependencies between the responsibilities, this set is then formed into a responsibility dependency graph. Furthermore, a domain-model for the system is given. The graph is encoded as a chromosome, which is then subjected to the genetic algorithm (implemented with Java). The algorithm transforms the CIM to a PIM through the implementation of a set of architectural patterns to the given model, and produces a UML class diagram as the result. Fig. 1 depicts the process from CIM to PIM in our approach.

3.1 Requirements model

Our CIM model consists of two parts, a responsibility graph and a domain model (see Fig. 1). A responsibility graph gives the functional requirements of the system in terms of responsibilities. A responsibility is either a task to be carried out by the system (or some part of it), or a data item that has to be managed by the system (or some part of it). Each node in the graph represents a responsibility, and each

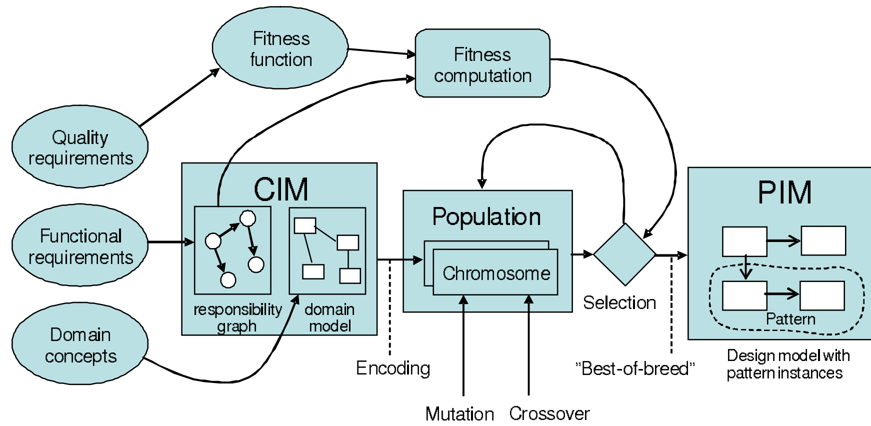


Fig. 1: Genetic model transformation process.

directed edge represents a dependency between the two responsibilities. Here a dependency means that the source responsibility relies on the target responsibility. In order to evaluate the system after the model has been subjected to transformation, some attributes of the responsibilities are also given, such as variability (the probability of future changes), parameter size and time consumption. The values for these attributes may be sometimes hard to determine at the requirements analysis stage, but the more accurately these can be estimated, the better will be the result. The given values for the attributes are relative, rather than absolute. The scales for the attributes are from 1 to 10 (parameter size, frequency, variability and call variability) or from 10 to 110 (execution time and call cost).

In our work, we have used an e-home as an example system. It contains 52 responsibilities and 90 dependencies between them. A part of the responsibility dependency graph, depicting the drape control component of the example system, is given in Fig. 2, where some sample properties (variability, parameter size, time consumption) are marked in the nodes. The drapeState node is marked with a thicker circle, as it is a data manager responsibility. The calculateOptimalDrape responsibility is a good example of how and why the certain attributes are evaluated: its variability is 3, as the optimal drape position can be calculated differently in different houses and according to different preferences. As it is a heavy operation, also its parameter size (6) and time consumption (90) are among the highest values of those shown here. Responsibilities with such high attribute values play an important role when constructing quality architecture as their placement has a bigger impact on the quality value.

The domain model part of CIM is constructed on the basis of the responsibility graph by assigning the responsibilities to classes. The division to classes is based on the data responsibilities: a responsibility is placed in the same class as the data it handles (either directly or through another responsibility, in the case where it does not use any data directly). The process actually follows the tradi-

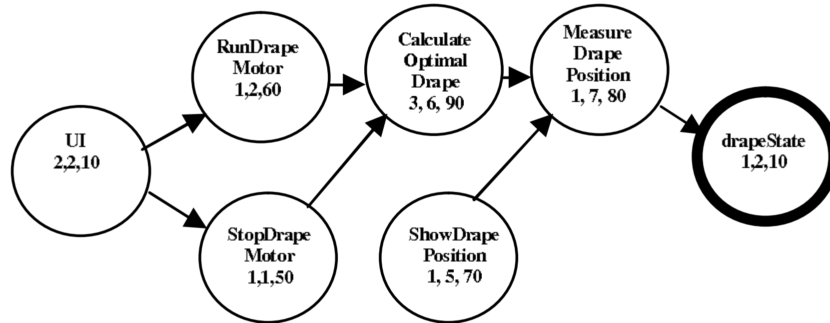


Fig. 2: A fragment of the responsibility dependency graph.

tional object-oriented method of extracting classes and their dependencies on the basis of a functional description of the system.

This initial model thus gives the system a basic structure by separating subsystems into components. Associations for the model are derived directly from the dependency graph. However, this structure may change, as the class division may be altered through application of architectural patterns.

The initial model is presented as a component diagram in Fig. 3. For simplicity, we have only used classes and associations. As can be seen, the initial model is very simplistic – the genetic algorithm is given a logical starting point with a preliminary decomposition. No decisions are made regarding the architectural styles or the finer details of the architecture.

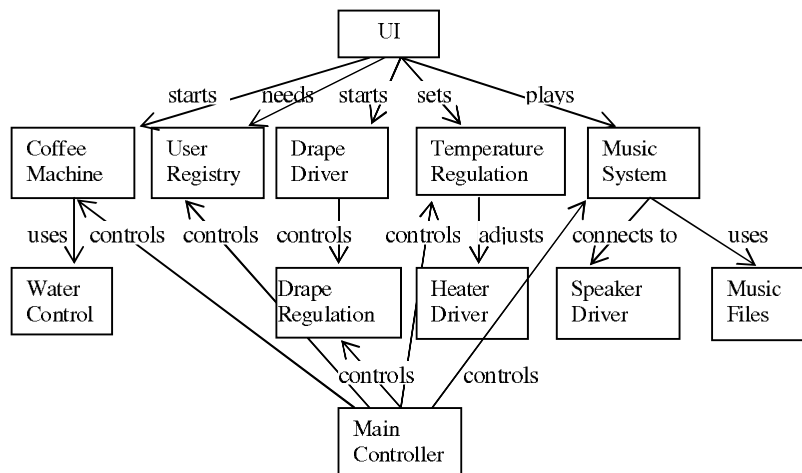


Fig. 3: Domain model of example system.

3.2 Genetic model representation

In a traditional chromosome representation, each chromosome consists of several genes, each of which has one field. A supergene, however, has several fields to store data in. Taking this idea as a starting point, it is quite straightforward to place all information regarding one responsibility into one supergene, i.e., each responsibility is represented as a supergene in the chromosome. This also makes it easier to keep the model consistent, as no responsibility can be left out of the model at any point, and there is no risk of breaking the dependencies between responsibilities.

There are two kinds of data regarding each responsibility r_i. Firstly, there is the basic information given as input. This contains the responsibilities $R_i = \{r_{i1}, r_{i2}, \ldots, r_{ik}\}$ depending on r_i, its name n_i, type d_i, frequency f_i, parameter size p_i, execution time t_i, call cost c_i, call variability cv_i and variability v_i. Secondly, there is the information regarding the responsibility r_i's place in the architecture: the class(es) $C_i = \{C_{i1}, C_{i2}, \ldots, C_{iv}\}$ it belongs to, the interface I_i it implements, the dispatcher D_i it uses, the responsibilities $RD_i \subset R_i$ that call it through the dispatcher, the design patterns $P_i = \{P_{i1}, P_{i2}, \ldots, P_{im}\}$ it is a part of and the predetermined model class MC_i. The dispatcher is given a separate field as opposed to other patterns for efficiency reasons. Fig. 4 depicts the structure of a supergene SG_i. The actual chromosome is formed by simply collecting all supergenes.

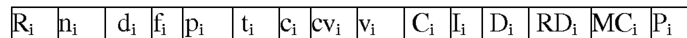


Fig. 4: Supergene.

Although the basic architecture is easy to keep coherent (i.e., the structure given by the model is not disturbed as the dependencies between responsibilities stay intact), the design patterns produce challenges at the chromosome level, as careless operations can easily break existing patterns and make the architecture incoherent. Thus, in order to quickly check the legality of an operation, a Pattern field is located in every supergene. This field contains information about the patterns with which the responsibility is involved, and thus if a new pattern is introduced, it is fast to check whether it contradicts the existing patterns. The initial population is made by first creating the desired number of individuals with the basic structure given in the CIM. A random pattern is then inserted into each individual, as a population should not consist of clones. In addition, a special case is left in the population where no pattern is initially inserted.

3.3 Mutation and crossover operations

As discussed, the model transformations are made by implementing patterns in the model. As we only deal with the refinements needed to transform a CIM to a PIM, we do not need to take into consideration the actual implementation of the patterns. The implementation is usually platform-specific, and thus belongs to

the PSM. The patterns we have chosen include very high-level architectural styles [19] (dispatcher and client-server), medium-level design patterns [6] (façade and mediator), and low-level design patterns [6] (strategy and proxy). This ensures a variety of "building material" for the genetic algorithm, and the different levels are also used with dynamic mutations at different stages of the development. The mutations are implemented in pairs of introducing a pattern or removing a pattern. The dispatcher architecture style makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the responsibilities can communicate through it. The mutations are the following:

- introduce/remove message dispatcher,
- communicate/remove communication through dispatcher,
- introduce/remove server,
- introduce/remove façade,
- introduce/remove mediator,
- introduce/remove strategy,
- introduce/remove proxy.

Apart from introducing the dispatcher, all the other mutations are implemented at supergene level by updating the information in the dispatcher connection or pattern field, as well as the interface and class fields when needed. However, as each pattern concerns more than one responsibility, all the supergenes involved in the implemented pattern must be mutated accordingly. For example, the proxy pattern is introduced by constructing a new ProxyPattern instance with data of the technical class and interface needed by the proxy. The pattern fields of the responsibility called through the proxy and the responsibilities calling the proxy are then updated so that they now contain the new ProxyPattern instance.

As adding a dispatcher, however, does not consider any particular responsibility (supergene), it cannot be implemented as a gene level mutation, but must affect the whole system. This means that the mutation must be implemented at chromosome level, where a dummy gene is added to the existing chromosome. In this gene, only the field containing information of the dispatcher has a value different from 0 or null. When a chromosome contains such a gene which "carries" the dispatcher, its supergenes can later on be subjected to the mutation where individual responsibilities are made to communicate through the dispatcher.

The crossover operation is implemented as a traditional one-point crossover. However, as patterns always concern more than one responsibility, it is possible that during a mutation or crossover, an existing pattern may be broken. Because of this, after each mutation and crossover operation, the resulting chromosome(s) are subjected to a corrective operation, which ensures that the model stays coherent. In the case of the crossover, it is decided that the left side of the offspring is always legal, and the adjustments are made to the right side of the crossover point. In addition to ensuring that the patterns present in the system stay coherent and "legal", the corrective function also checks that the model conforms to certain architectural laws. These laws demand uniform calls between two classes (e.g., through an interface or a dispatcher), and state some basic rules regarding architectures, e.g., no

responsibility can implement more than one interface. These laws ensure that no anomalies are brought to the model.

Mutation and crossover indexes are chosen randomly. An exception is made in case of patterns which favor highly variable responsibilities (dispatcher communication and strategy) and proxy, which favors responsibilities that have a relatively high call variability value. In these cases, the chances of each particular responsibility to be subjected to such mutations increases linearly in relation to the respective property value, e.g., if responsibility r_i has a variability value 3, it is three times more likely to be subjected to a strategy or dispatcher mutation than responsibility r_j with variability value 1.

The actual mutation probabilities are given as input. Selecting the mutation is made with a "roulette wheel" selection [11], where the size of each slice of the wheel is in proportion to the given probability of the respective mutation. Null mutation and crossover are also included in the wheel. Crossover is treated as a mutation in the selection, as the probability of being subjected to crossover increases in relation to how high an individual's fitness is in the population. When the crossover is included in the same wheel as the mutations, it also leads to smaller mutation slices, as they need to accommodate to fit in the larger crossover slice. Thus, including crossover and mutation in the same wheel increases the chances that highly fit individuals remain untouched throughout generations. Each individual has a chance of reproducing in each generation: if the first roulette selection lands on a mutation, another selection is performed after the mutation has been administered. If the second selection lands on the crossover slice, the individual may produce offspring. In any other case, the second selection is not taken into account, i.e., the individual is not mutated twice.

We have also adopted the idea of dynamic mutation probabilities. The three different levels of patterns can be further used in mutation probabilities by favoring the high-level patterns in the beginning of evolution, as the basic structure of the model is being defined. We then move on to favor the medium-level patterns, and in the very end of evolution, the low-level patterns are favored to ensure as high-tuned a solution as possible. In practice, the probabilities of high-level pattern introductions are decreased after 1/3 of the generations have passed, and the probabilities of medium-level pattern introductions are increased respectively. The same procedure is done between medium-level pattern introductions and low-level pattern introductions after another 1/3 of the generations.

3.4 Fitness function

Selecting an appropriate fitness function is probably the most demanding task with any genetic algorithm application when there is no clear value to measure from the solutions. In the case of software architecture, evaluation is even more difficult. In real world, evaluation of software architecture is almost always done manually by human designers, and metric calculations are only used as guidelines. Also, no two architects may ever agree on a unique quality for certain architecture, as evaluation is bound to be subjective, and different values and backgrounds will influence the outcome of any evaluation process. However, for the genetic algorithm to be able

to evaluate the architecture, a purely numerical fitness value must be calculated.

In a fully automated approach, no human interception is allowed, and thus the fitness function needs to be based on metrics. The selection of metrics may be as arguable as the evaluations of two different architects on a single software architecture. The reasoning behind the selected metrics in this approach is that they have been widely used and recognized to accurately measure some quality aspects of software architecture. Hence, the metrics are chosen so that they measure quality aspects that can be seen as ”most agreed upon” in the real world, and singular values can be seen as accurate as possible. However, the combination of metrics and multiple optimization is another problem entirely, as not many quality values can be optimized simultaneously.

The fitness function is based on software product metrics, most of which are from the metrics suite introduced by Chidamber and Kemerer [3]. These metrics, especially coupling and cohesion, have been used as a starting point for the fitness function, and have been further developed and grouped to achieve clear ”sub-fitnesses” for modifiability and efficiency, both of which are measured with a positive and negative metric. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-server architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. The inefficiency caused by patterns may be ameliorated by clever compilers, but as we are interested in the software architecture at a more high level, we do not consider the effect of compilers here. A complexity metric is added to penalize having many classes and interfaces as well as extremely large classes.

Dividing the fitness function into sub-functions answers the demands of the real world. Hardly any architecture can be optimized from all quality viewpoints, but some viewpoints are ranked higher than others, depending on the demands regarding the architecture. By separating efficiency and modifiability, which are especially difficult to optimize simultaneously, we can assign a bigger weight to the more desired quality aspect. When w_i is the weight for the respective sub-fitness sf_i, the fitness function $f(x)$ for chromosome x can be expressed as

$$f(x) = w_1 \times \mathrm{sf}_1 - w_2 \times \mathrm{sf}_2 + w_3 \times \mathrm{sf}_3 - w_4 \times \mathrm{sf}_4 - w_5 \times \mathrm{sf}_5.$$

Here, sf_1 measures positive modifiability, sf_2 negative modifiability, sf_3 positive efficiency, sf_4 negative efficiency and sf_5 complexity. The sub-fitness functions are defined as follows ($| X |$ denotes the cardinality of X):

$$\begin{aligned}\mathrm{sf}_1 = {} & |\text{ interface implementors }| + |\text{ calls to interfaces }| + \\ & (|\text{ calls through dispatcher }| \times \\ & \textstyle\sum(\text{variabilities of responsibilities called through dispatcher})) - \\ & |\text{ unused responsibilities in interfaces }| \times 10,\end{aligned}$$

$$\begin{aligned}\mathrm{sf}_2 = {} & |\text{ calls between responsibilities in different classes that do not} \\ & \text{happen through a pattern }|,\end{aligned}$$

$$
\begin{aligned}
\mathrm{sf}_3 \quad = \quad & \sum(\mid \text{dependingResponsibilities within same class} \mid \times \\
& \text{parameterSize} + \\
& \sum(\mid \text{usedResponsibilities in same class} \mid \times \text{parameterSize} + \\
& \mid \text{dependingResponsibilities in same class} \mid \times \text{parameterSize})),
\end{aligned}
$$

$$
\mathrm{sf}_4 = \sum \text{ClassInstabilities} + (\mid \text{dispatcherCalls} \mid + \mid \text{serverCalls} \mid) \times \sum \text{callCosts},
$$

$$
\mathrm{sf}_5 = \mid \text{classes} \mid + \mid \text{interfaces} \mid .
$$

The multiplier 10 in sf_1 means that having unused responsibilities in an interface is almost an architecture law, and should be more heavily penalized.

3.5 Selection

A selection operation is needed as the size of the population should be the same at the start of each generation, but through crossover the amount of individuals grows. Selecting the individuals for each generation is made with the roulette wheel method. Each individual is given a ”slice” on the wheel. The size of the ”slice” is based on how high the individual’s fitness is in the population. Thus, the ”slices” are not proportioned according to raw numerical fitness differences, but based on order of fitnesses. In addition to roulette wheel selection an elitist approach is used, where 10 individuals with the best fitnesses are kept after each generation.

This approach takes into consideration that the quality difference between two individuals may not be the same as would appear when examining the raw numerical difference between the fitness values. By using elitism as an addition to the roulette wheel selection, the development of fitness values is also more secured.

4. Experiments

In this section we present the results achieved through experiments with our approach. The consistency of the fitness development of each test run has been showed in our previous research [15], and thus the fitness curves presented here can reliably be calculated as averages. We have first calculated the average fitness of the 10 best individuals of each generation, thus achieving the fitness development curve of the fittest individuals in each run. The actual fitness curves are then achieved by calculating the average development of five test runs. The mutation probabilities used for the test runs are same in all the experiments discussed here, and were set after exhaustive testing.

As using a model to give a basic structure for each solution greatly affects the amount of variation between individuals, we first experimented how the population size affected the fitness curve. As can be seen in Fig. 5, with a population of 50 the fitness curve actually turns downwards already after some 200 generations,

while with populations of 100 and 150 the development is more stable. Thus the forthcoming tests were made with a population of 100, as it ensured a further developing fitness curve by having more variability in the population. Increasing the population size even further to 150 did not seem necessary, as the fitness values were not noticeably better than those achieved with a population of 100.

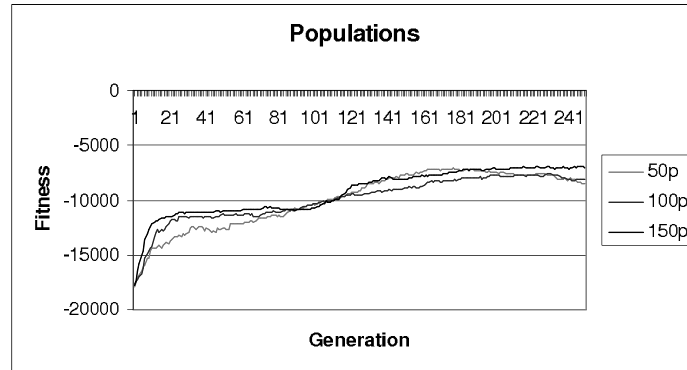


Fig. 5: Different population sizes.

Another common variable to all genetic algorithms in addition to population size is the number of generations the algorithm runs through. We tested our algorithm with 1000 generations, and the resulting fitness curve is depicted in Fig. 6. As can be seen, the fitness curve continues to steadily achieve higher values up until around 625 generations, after which its development plunges. This would indicate that running the algorithm with an exceptionally large number of generations would not be beneficial. The following tests are all made with either 250 or 500 generations, when we can be sure that the fitness curve has not passed its optimum and started to descend.

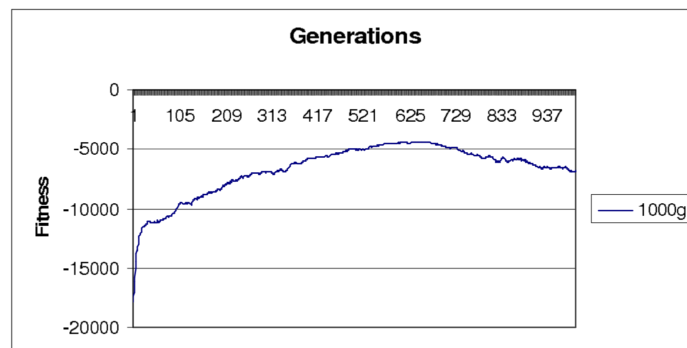


Fig. 6: 1000 generations.

In Figures 5 and 6 the fitness curve indicates the development of the overall fitness, i.e., modifiability, efficiency and complexity combined. As has been discussed in Section 3, balancing the different sub-fitnesses is a demanding optimization task, and thus it is interesting to see how weighting one quality attribute over another will affect the development of their individual fitness curves. In Fig. 7 we have separated the fitness curves for modifiability and efficiency from the overall fitness.

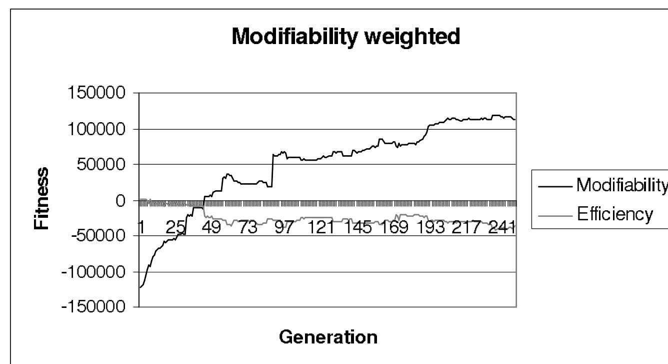


Fig. 7: Modifiability weighted.

In this test, modifiability was valued 10 times higher than efficiency, which results in the modifiability curve increasing quite rapidly, while the efficiency curve does not develop at all, but achieves quite low values throughout the generations. This is expected, as when modifiability is valued, solutions with high efficiency values do not survive to next generation, and are definitely not in the top of any generation. A similar test was also made where efficiency was valued 10 times higher than modifiability; the fitness curves for this experiment are shown in Fig. 8.

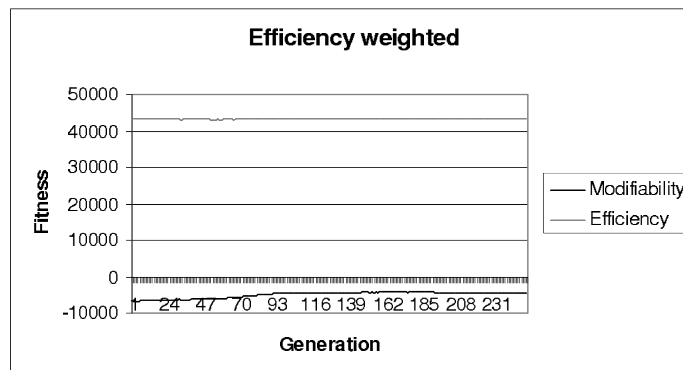


Fig. 8: Efficiency weighted.

The fitness curves achieved while valuing efficiency are quite different from those achieved when weighting modifiability: in the case of efficiency, neither the efficiency nor the modifiability curve develops at all. Valuing efficiency can only be seen in the different ranges of values, as the efficiency curve achieves high positive values while the modifiability curve never goes above 0. The reason for the stable efficiency curve can be found in the initial population: as the initial population only contains one pattern per model, and even a solution where there are no patterns, these models achieve such high efficiency values that it is simply not possible to top them as each time a pattern is added, the efficiency of the solution decreases.

In addition to studying the effects of weighting the different quality attributes, we have studied the effect of the dynamic mutation probabilities, discussed in Section 3, to the fitness curve. Fitness curves achieved with dynamic and static mutation probabilities, when both quality attributes are valued equally high, are shown in Fig. 9. The development of the overall fitness is quite similar, and the tests with static mutations actually achieve slightly higher fitness values toward the end.

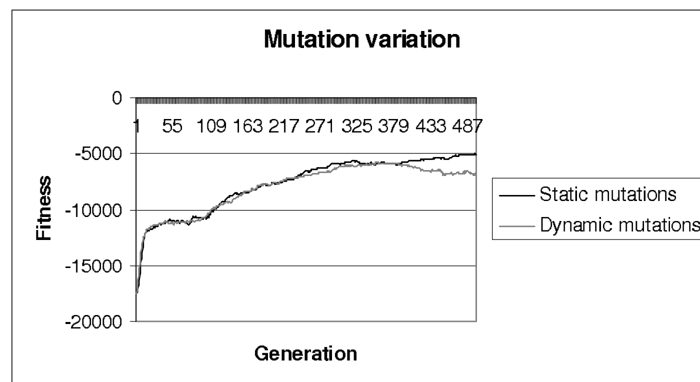


Fig. 9: Fitness curves with static and dynamic mutation probabilities.

However, when modifiability is weighted over efficiency, we can see that the modifiability fitness clearly benefits from the dynamic mutation probabilities, as shown in the curve in Fig. 10, where the modifiability sub-fitness curves are shown from experiments with dynamic and static mutation probabilities.

As these experiments show, the modifiability of the solutions can quite straightforwardly be affected by weighting the modifiability sub-fitness functions over the efficiency sub-fitnesses. The implemented dynamic mutation probabilities also benefit the modifiability fitness, which makes their usage justified.

The achieved solutions illustrate the creative power of the genetic algorithm: there is a high degree of variability between the solutions, and different patterns are implemented fearlessly. However, although some good decisions are found in parts of each solution, no single solution is achieved that would instantly impress an expert as having overall top quality.

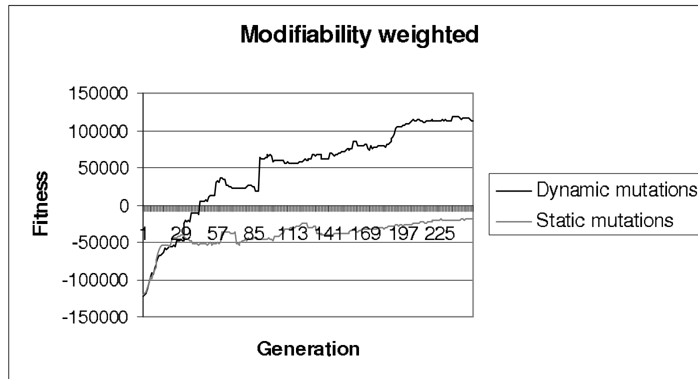


Fig. 10: Modifiability curves with dynamic and static mutation probabilities when modifiability weighted over efficiency.

Fig. 11 illustrates a proposed solution. In this solution, the dispatcher is central as many components communicate through it. A large component is turned into a server, and the user interface further uses a proxy to call the responsibilities provided by the server. Strategies are also used for several highly variable responsibilities (the responsibility and its class are given in parentheses).

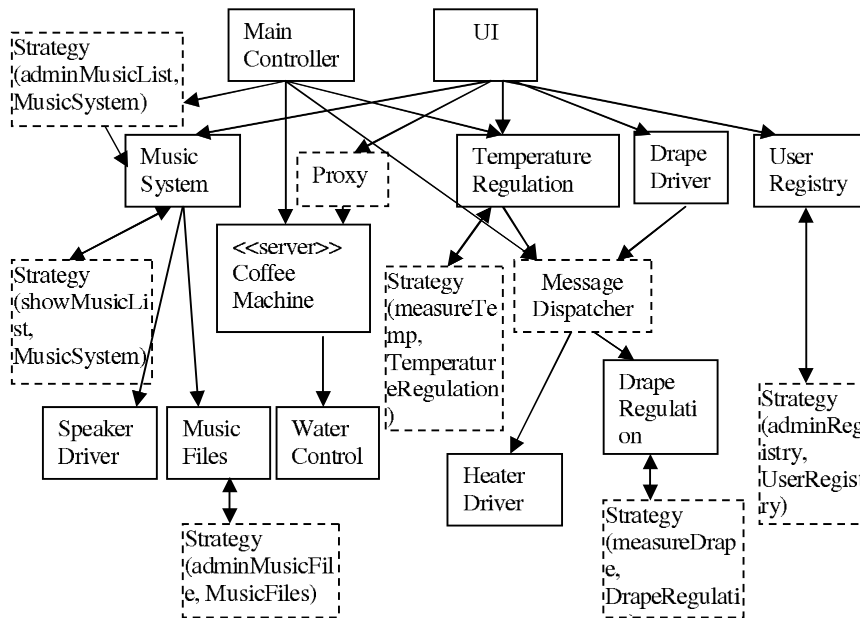


Fig. 11: Generated PIM (abstracted from class diagram).

5. Conclusions

We have presented a novel approach for model transformations from CIM to PIM using a genetic algorithm that implements design patterns to the given domain model and gives a solution model as a UML class diagram. Our work differs from that of Amoui *et al.* [2] by using the model-driven approach, not having a ready-made architecture to start from and not storing information of used patterns as a sequence of transformations.

The results show that promising solutions can be achieved, as the quality of solutions improves over generations and the actual design choices seen in the class diagrams are sensible in many cases. The achieved solutions prove that a conceivable PIM level architecture can be achieved from very high-level requirements, when only given a domain model as a guideline of the architecture's structure.

The biggest limitations in our approach at the moment are the small number of applied patterns and the inconsistency of quality in proposed solutions. The inconsistency of quality is most likely due to the difficulty of finding a suitable compromise between the selected quality estimators. Also, the implementation of dynamic mutations should be studied further, as it may also have a negative impact if such design patterns that are not possible to implement in the architecture are favored for a long time.

6. Future work

Our plans for future work include implementing a simulated annealing algorithm for comparison, a multi-objective fitness function to achieve a more balanced architecture in terms of different quality values, and studying different ways of applying the dynamic mutations.

Implementing the simulated annealing algorithm will show whether other search-based techniques could also be used for model transformations, or whether the genetic algorithm is the only one that is "sophisticated enough" to handle the intricate details regarding software architectures. If the simulated annealing approach provides good results, it may be considered whether a combination of the two algorithms could be used to achieve optimal results.

Presumably, a multi-objective fitness function will better address the problem of evaluating a software architecture. As discussed, it is difficult to optimize several quality aspects at the same time. A multi-objective fitness function will enable to better choose a solution that achieves the best compromise between different quality measures. In addition, while further improving mutations and the fitness evaluation will most probably improve the quality of results, more design patterns are also to be added in order to gain access to a wider collection of design choices.

References

[1] A , A., K , G., S , F. 2003. Graph Transformations on Domain-Specific Models. Technical Report ISIS-03-403, Institute for Software Integrated Systems Vanderbilt University.

[2] A , M., M , S., A , S., L , C. 2006. A Genetic Algorithm Approach to Design Evolution using Design Pattern Transformation. *International Journal of Information Technology and Intelligent Computing 1*, 235–245.

[3] C , S.R. K , C.F. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering 20*, 6, 476–492.

[4] C , J., D , J.J., H , M., H , R.M., J , B., L , M., M , B., M , S., R ., K., R , M., S , M. 2003. Reformulating Software Engineering as a Search Problem. In *IEE Proceedings - Software*, Volume 150, 161–175.

[5] D , S., G , C., S , S. 2005. Model transformation with the IBM model transformation framework. Report, IBM, http://www-128.ibm.com/developerworks/.

[6] G , E., H , R., J , R., V , J. 1995. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[7] H , M., H , R., P , M. 2000. A New Representation and Crossover Operator for Search-based Optimization of Software Modularization. In *Proceedings of GECCO'00*, 1351–1358.

[8] K , A., B , J., C , E. 2004. Model Transformation Language MOLA. In *Proceedings of MDAFA*, 14–28.

[9] K , N. 2006. Transformation techniques in the model-driven development process of UWE.

[10] L , C., G , J., J , J., S , T. 2007. Applying Triple Graph Grammars for Pattern-based Workflow Model Transformations. *Journal of Object Technology 6*, 9, 253–273.

[11] M , Z. 1992. *Genetic Algorithms + Data Structures = Evolutionary Programs*. Springer.

[12] O'K , M. C , M. 2007. Getting the most from search-based refactoring. In *Proceedings of GECCO'07*, 1114–1120.

[13] OMG. 2003. Technical Guide to Model Driven Architecture: The MDA Guide v1.0.1. Report, www.omg.org.

[14] OMG. 2005. MOF QVT Final Adopted Specification. Report, www.omg.org.

[15] R , O., K , K., M , E. 2008. Genetic Synthesis of Software Architecture. In *Proceedings of SEAL'08*. Springer LNCS, 565–574.

[16] R , A, F -M , E., P , M. 2008. CIM to PIM Transformation: A Reality. In *Research and Practical Issues of Enterprise Information Systems*, Volume II, 1239–1249.

[17] S , A. 1994. Specification of Graph Translators with Triple Graph Grammars. In *Proceedings of the 20 International Workshop on Graph-Theoretic Concepts in Computer Science*, 151–163.

[18] S , O., S , J., B , D. 2006. Search-based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems. In *Proceedings of GECCO'06*, 1909–1916.

[19] S , M. G , D. 1996. *Software Architecture - Perspectives on an Emerging Discipline*. Prentice-Hall.

[20] V , D., V , G., P , A. 2005. Designing the Automatic Transformation of Visual Languages. *Science of Computer Programming 44*, 2, 205–227.

[21] W3C. 1999. XSL Transformations (XSLT) Version 1.0. Report, http://www.w3.org/.

[22] Z , W., M , H., Z , H., Y , J. 2005. Transformation from CIM to PIM: A Feature-oriented Component-based Approach. In *Proceedings of Models. LNCS 3713*. Springer, 246–263.

Paper III

Outi Räihä, Kai Koskimies and Erkki Mäkinen, Scenario-Based Genetic Synthesis of Software Architecture, In: *Proc. of the 4th International Conference on Software Engineering Advances (ICSEA'09)*, Porto, Portugal. September 2009, IEEE Press, 437-445.

© 2009 IEEE. Reprinted, with permission.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University'of Tampere's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this material, you agree to all provisions of the copyright laws protecting it.

2009 Fourth International Conference on Software Engineering Advances

Scenario-Based Genetic Synthesis of Software Architecture

Outi Räihä, Kai Koskimies
Department of Software Systems
Tampere University of Technology
Tampere, Finland
outi.raiha@tut.fi, kai.koskimies@tut.fi

Erkki Mäkinen
Department of Computer Sciences
University of Tampere
Tampere, Finland
em@cs.uta.fi

Abstract— **Software architecture design can be regarded as finding an optimal combination of known general solutions and architectural knowledge with respect to given requirements. Based on previous work on synthesizing software architecture using genetic algorithms, we propose a refined fitness function for assessing software architecture in genetic synthesis, taking into account the specific anticipated needs of the software system under design. Inspired by real life architecture evaluation methods, the refined fitness function employs scenarios, specific situations possibly occurring during the lifetime of the system and requiring certain modifiability properties of the system. Empirical studies based on two example systems suggest that using this kind of fitness function significantly improves the quality of the resulting architecture.**

Keywords- software architecture; software design; genetic algorithm; search-based software engineering

I. INTRODUCTION

Design of software architecture is one of the most critical and intellectually demanding activities of software engineering. Often a software architect has a limited mindset influenced by her prior experiences in a particular domain. Although a vast amount of architectural knowledge exists in the form of documented, general "good" solutions (like design patterns [5]), it is difficult for a human architect to come up with an optimal combination of such solutions. Architects typically apply a limited set of solutions they know have worked in their previous systems. Unfortunately, this often leads to costly iteration of the architectural design, due to quality-related problems observed during implementation, testing, and usage.

In our previous work [11, 12] we have taken the viewpoint that the design of software architecture is essentially a combinatorial task where the problem is to find an optimal conformation of known good practices and solutions of architectural design in the context of a particular application. An optimal software architecture is the one which supports the realization of the quality requirements of the system under design as well as possible. Assuming that both the architecture and the general solutions can be formally represented, and that the quality of an architecture can be effectively measured, the problem of architecture design becomes a search problem that is appropriate for a heuristic search algorithm.

Although fully automated software architecture design seems unrealistic in practice, we argue that an architect can significantly benefit from an automatically synthesized, unbiased architecture proposal exploiting a wide knowledge base of architectural solutions and practices. We expect that such a proposal can be used as a starting point for human software architecture design, speeding up the design process and improving the quality of the resulting architecture.

We have shown [11, 12] that genetic algorithms can be applied to the problem of architectural design, interpreting general solutions as mutations and using architectural metrics as a basis for the fitness function. The quality of an architecture was measured with respect to three quality attributes: modifiability, efficiency and complexity. The architecture objects were presented as UML class diagrams. The input of the genetic algorithm consisted of a responsibility graph, that is, a graph of tasks the application is supposed to be able to perform. This corresponds to the functional requirements of the application. In addition, a domain model of the application was given as an initial rudimentary architecture. We will explain this technique in more detail in Section III.

Although this technique produced reasonably good architectures in terms of general metrics, the resulting architectures were typically unsatisfactory from the viewpoint of the anticipated specific needs of the particular target application. This concerned in particular modifiability: the architectures exhibited good modifiability characteristics, but not necessarily in a way that would support the most likely modifications during the lifetime of the target system. It seemed obvious that the fitness function must be tailored for a particular target system to improve the results.

In this paper we introduce a technique to take into account the specific expected modifiability requirements of a system in the genetic synthesis of its software architecture. The idea is based on using *scenarios*, that is, concrete imaginary situations that are expected to take place during the evolution of the system. Each scenario specifies a particular modification need and its probability, and the fitness function computes the suitability of the architecture for the scenarios, in addition to the general metrics. This technique resembles – and is inspired by – the way software architectures are evaluated in real life: for example, ATAM [7] is a popular software architecture evaluation method where different stakeholders invent scenarios that are analyzed against the architectural solutions. We show that this kind of refined fitness function leads to improved

978-0-7695-3777-1/09 $26.00 © 2009 IEEE
DOI 10.1109/ICSEA.2009.70

IEEE computer society

development of the quality of the architecture during the genetic synthesis.

The paper is organized as follows. In the next section we will briefly discuss genetic algorithms, their use in software engineering and different approaches to define the fitness function. In Section III we summarize our basic approach for genetic synthesis of software architecture. Section IV discusses the idea of scenario-based fitness evaluation in more detail. We have carried out a series of experiments to study the effect of this technique. The results of these experiments are discussed in Section V. Finally, concluding remarks and ideas on future research directions are presented in Section VI.

II. Related Work

A genetic algorithm maintains a population of possible solutions (individuals). In our problem a population is a set of possible architectures which undergoes an evolutionary process imitating the natural biological evolution. In each generation better individuals have greater possibilities to survive and reproduce, while worse individuals have greater possibilities to die and to be replaced. It is believed that this process leads to a combination of the properties of the better individuals, which constitutes a good solution to the problem in question.

To operate with a genetic algorithm, one needs an encoding of possible solutions, an initial population, mutation and crossover operators, a fitness function and a selection operator for choosing the survivors for the next generation. We do not explain these concepts in detail here; we assume that the reader is familiar with the basics of genetic algorithms, as given e.g. by Michalewicz [8].

Search-based software engineering has previously focused on separate subproblems, such as software clustering and refactoring, see, e.g., [4, 6]. For a thorough survey on search-based software design, see [10]. Recently, approaches dealing with higher level structural units, such as design patterns, have gained more interest. For example, Amoui et al. [1] apply genetic algorithms for finding the optimal sequence of design pattern transformations to increase the reusability of a software system, and Simons and Parmee [15, 16] take use cases as the starting point for system specification.

Räihä et al. [11] start the genetic design of a software architecture from a responsibility dependency graph. In this solution, each responsibility is represented by a supergene, and a chromosome is a collection of supergenes. The supergene contains information regarding the responsibility, such as dependencies of other responsibilities, and attributes such as estimated time consumption. Mutations are implemented as splitting/joining class(es) or adding/removing generic architectural solutions which may be design patterns [5], architectural styles [14] or interfaces.

In the work of Räihä et al. [11], the fitness function is a combination of object-oriented software metrics, which have been grouped to measure efficiency and modifiability. Furthermore, a complexity measure is introduced to evaluate the overall understandability of the architecture. The fitness function is then defined as a weighted sum of these factors.

Räihä et al. [12] also apply genetic algorithms in model transformations that can be understood as pattern-based refinements, using design patterns as the basis of mutations. The fitness function was similar to the one in the previous study.

However, tests have shown [13] that the fitness function based on traditional quality metrics does not fully satisfy the needs in the search process of a software architecture. To this end, the present paper aims at a more accurate fitness function, which is tailored for the design process at hand. While the previous fitness functions [11, 12] measure generally defined properties of a software architecture, we now try to apply measurements that take into account more detailed exigencies of the environment where the architecture must “survive”. Biologically thinking, we can say that the scenario-based fitness function plays the role of a specific environment to which the population must adapt.

III. Genetic Architecture Synthesis

In this section we describe in more detail our basic approach to genetically synthesize software architecture design [11, 12]. This approach has been realized in Java and experimented with the same sample systems as here.

A. *Requirements*

For any system, the functionality can be described at a very high level by identifying the various functional responsibilities the system is supposed to provide, and their mutual dependencies. For example, consider a system controlling a computerized home, e-home. This system obviously has responsibilities like “playing music” or “making coffee”. The former responsibility requires more fine-grained responsibilities like “start music playing” and “stop music playing”. Furthermore, the system needs to manage certain data items, like musical piece data, suggesting data management responsibilities. Identifying the responsibilities in this way and analyzing their mutual dependency relationships leads to a responsibility graph, where each node represents a responsibility, and each directed edge represents a dependency between the two responsibilities (that is, the source responsibility requires the target responsibility).

In order to facilitate the evaluation of the architecture of the system, the responsibilities can be associated with qualifying attributes, such as sensitiveness to variation in implementation and time consumption. The values for these attributes of course cannot be known precisely for a system under design, but estimations of these (relative) values provide more information for the genetic algorithm, regarding modifiability and efficiency fitness.

We have used two sample systems in our experiments. In addition to the e-home control system discussed above, we have applied our technique to synthesize an architecture for a robot war simulation application (called robowar hereafter). The latter system is a computer game where the players can give different characteristics to war robots and observe how they perform. We wanted to experiment with systems that

have fairly different characteristics, e-home being a typical embedded system and robowar a typical framework-based game system. The e-home system [9] yielded 52 responsibilities and 90 dependencies between the responsibilities, while robowar has 56 responsibilities and 73 dependencies between them. A part of the e-home responsibility dependency graph is given in Figure 1. The nodes representing data manager responsibilities are marked with a thick circle.

In addition to simply giving the responsibilities and their relationships, we have built a domain model for both applications that contains the logical entities appearing in the responsibilities, and the relationships between these entities.

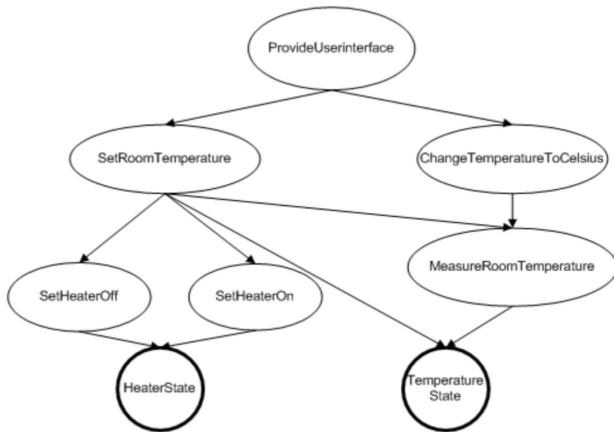


Figure 1. A fragment of the responsibility graph for e-home

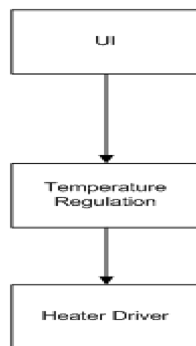


Figure 2. A fragment of the domain model for e-home

A fragment of the e-home domain model is given in Figure 2; this corresponds to the responsibility graph fragment of Figure 1. As can be seen, the initial model is very simplistic – the genetic algorithm is merely given a logical starting point with preliminary conceptual decomposition. No decisions are made regarding the actual architectural solutions introduced later as mutations. The entities (classes) in the domain model are identified by studying the names of the responsibilities. Each responsibility is assigned to a single class, and the associations between the classes are derived from the dependencies between the responsibilities.

The actual input for the genetic algorithm is thus a combination of the responsibility dependency graph and the domain model, resulting in a simplistic class diagram. This initial model gives the system a basic decomposition. However, this structure is not frozen, as the class division may be altered through application of architectural patterns.

B. Genetic Representation

The genetic algorithm operates with two kinds of data regarding each responsibility. Firstly, the basic input contains the other responsibilities depending on the responsibility, other attributes like estimated parameter size, execution time and variability sensitiveness, and the domain model class it is initially placed in. Secondly, there is the information regarding the responsibility's place in the architecture: the class(es) it belongs to, the interface it implements, the dispatcher (see below) it uses, the responsibilities that call it through the dispatcher, and the design patterns and styles (apart from dispatcher) it is a part of. The dispatcher is given a separate field as opposed to other patterns for efficiency reasons. The latter data is produced by the genetic algorithm. All data regarding a responsibility is encoded as a supergene. The chromosome handled by the genetic algorithm is gained by collecting the supergenes, i.e., all data regarding all responsibilities, thus representing a whole view of the architecture.

The initial population is made by first creating the desired number of individuals with the basic structure given in the responsibility dependency graph and domain model. A random pattern is then inserted into each individual, as an initial population should not consist entirely of clones. In addition, a special individual is left in the population where no pattern is initially inserted: this ensures versatility in the population.

C. Mutation and Crossover Operations

As discussed above, the actual design is made by adding patterns to the system. The patterns have been chosen so that there are very high-level architectural styles [14] (dispatcher and client-server), medium-level design patterns [5] (Façade and Mediator), and low-level design patterns [5] (Strategy, Adapter and Template). The mutations are implemented in pairs of introducing a pattern or removing a pattern [9]. The dispatcher architecture style [14] makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the responsibilities can communicate through it. The mutations are the following:

- introduce/remove message dispatcher
- create link/remove link to dispatcher
- introduce/remove server
- introduce/remove façade
- introduce/remove mediator
- introduce/remove strategy
- introduce/remove adapter
- introduce/remove template.

The crossover operation is implemented as a traditional one-point crossover with a corrective operation. This operation ensures that the architecture stays coherent, as

patterns may be broken by overlapping mutations. In addition to ensuring that the patterns present in the system stay coherent and “legal”, the corrective function also checks that the design conforms to certain architectural laws. These laws demand uniform calls between two classes (e.g., through an interface or a dispatcher), and state some basic rules regarding architectures, e.g., no responsibility can implement more than one interface. The purpose of these laws is to ensure that no anomalies are brought to the design.

The actual mutation probabilities are given as input. Selecting the mutation is made with a “roulette wheel” selection [8], where the size of each slice of the wheel is in proportion to the given probability of the respective mutation. Null mutation and crossover are also included in the wheel. Each individual has a chance of reproducing in each generation: if the first roulette selection lands on a mutation, another selection is performed after the mutation has been administered. If the second selection lands on the crossover slice, the individual may produce offspring. In any other case, the second selection is not taken into account, i.e., the individual is not mutated twice.

D. Core Fitness Function and Selection

The core fitness function is based on widely used software product metrics, most of which are from the metrics suite introduced by Chidamber and Kemerer [3]. These metrics, especially coupling and cohesion, have been used as a starting point for the core fitness function, and have been further developed and grouped to achieve clear “sub-fitnesses” for modifiability and efficiency, both of which are measured with a positive and negative sub-function. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-sever architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. A complexity metric is added to penalize having many classes and interfaces as well as extremely large classes.

Dividing the core fitness function into sub-functions answers the demands of the real world. Hardly any architecture can be optimized from all quality viewpoints, but some viewpoints are ranked higher than others, depending on the demands regarding the specific architecture at hand. By separating efficiency and modifiability, which are especially difficult to optimize simultaneously, we can assign a bigger weight to the more desired quality aspect. When w_i is the weight for the respective sub-fitness sf_i, the core fitness function $f_c(x)$ for chromosome x can be expressed as

$$f_c(x) = w_1*sf_1 - w_2*sf_2 + w_3*sf_3 - w_4*sf_4 - w_5*sf_5.$$

Here, sf_1 measures positive modifiability, sf_2 negative modifiability, sf_3 positive efficiency, sf_4 negative efficiency and sf_5 complexity. The sub-fitness functions are defined as follows (|X| denotes the cardinality of X):

sf_1 = |interface implementors| + |calls to interfaces| + (|calls through dispatcher| ∗ ∑ (variabilities of responsibilities called through dispatcher)) – |unused responsibilities in interfaces| ∗ α,

sf_2 = |calls between responsibilities in different classes, that do not happen through a pattern|,

sf_3 = ∑ (|dependingResponsibilities within same class| ∗ parameterSize + ∑ |usedResponsibilities in same class| ∗ parameterSize + |dependingResponsibilities in same class| ∗ parameterSize)),

sf_4 = ∑ ClassInstabilities + (|dispatcherCalls| + |serverCalls|)∗ ∑ callCosts, and

sf_5 = |classes| + |interfaces|.

The multiplier α in sf_1 emphasizes that having unused responsibilities in an interface is almost an architecture law, and should be more heavily penalized.

Selecting the individuals for each generation is made with the same kind of roulette wheel method as was used for choosing the mutations. Such a selection operation is needed, as the size of the population should be the same at the start of each generation, but through crossover the amount of individuals grows.

IV. Using Scenarios in Fitness Function

A. Defining scenarios

Basically, a scenario describes an interaction between a stakeholder and the system [2]. Scenarios are typically used as test cases for analyzing the architecture from the viewpoint of some quality attribute. For example, a scalability scenario could describe a situation where a certain number of users are accessing the system at the same time in a certain way, stressing the scalability of the system for a large number of users. In scenario-based architectural assessment the stakeholders invent and prioritize scenarios, and the software architect explains how the architecture is supposed to handle the scenario. Scenarios which get weak support in the architecture are classified as risks, corresponding to failures in traditional testing.

In our approach we have used scenarios that stress the modifiability of the system. When studying scenarios from an actual architecture evaluation document, we noticed that scenarios relating to modifiability can be roughly divided into two categories: adding an alternative version for a component or changing a component. In addition, we could divide scenarios according to whether the modification should be done statically or dynamically, and whether a change concerns only the implementation of a component or the entire meaning of the component. Thus each scenario contains the following information: the responsibility it involves, whether it requires addition or change, whether it is static or dynamic and is the change semantic or implementational. In addition, we used probabilities to describe how likely it is that the scenario takes place. This can be also seen as an indicator of the general importance of the scenario.

B. Examples

For both sample systems (e-home and robowar) 15 different modifiability scenarios were invented based on the assumed evolution of these kinds of systems.

For example, scenarios for the robowar system include:

- the player should be able to change the appearance of a robot (scenario probability 80%)
- the developer should be able to add an alternative armor control (80%)
- the developer should be able to change the implementation of the robot intelligence control (90%).

Scenarios for the e-home system include:

- the user should be able to change the way the music list is showed (90%)
- the developer should be able to change the way water is connected to the coffee machine (50%)
- the developer should be able to add another way of showing the coffee machine status (60%).

These scenarios are then analyzed to fit the classification framework of scenarios discussed above. The first robowar scenario is a changing scenario that should be handled dynamically. It concerns implementation. The second robowar scenario is an adding scenario that should be handled statically and again handles implementation. The last robowar scenario is a changing scenario to be implemented statically.

The first e-home scenario is a changing scenario that regards implementation and should be handled dynamically. The second e-home scenario is a changing scenario that should be handled statically and concerns the semantics of the component, i.e., the interface may need to be changed. The final e-home scenario is a static adding scenario concerning implementation.

C. Using scenarios as a fitness function

In order to use the verbally defined scenarios in a fitness function, they need to be translated so that a numerical value is achieved from evaluating how the proposed architecture handles a scenario.

We begin by encoding each scenario according to the five classification criteria discussed in Subsection IV.B: involved responsibility, adding/changing, static/dynamic, semantic/implementational, probability. After the scenarios have been encoded, they are given to a scenario interpreter. This contains information of what kind of solutions generally work for a certain type of scenario. For example, if the implementation of a responsibility should be statically changed, it is commonly known that a Template Method pattern [5] might be a good solution in that situation. Note that no explicit rules regarding the structure of the sample systems are given, only common knowledge about how certain structural solutions (e.g. design patterns, interfaces) support modifications. Different solutions are given an order of preference with regard to the type of the scenario at hand.

When the genetic algorithm calculates a scenario fitness value, it goes through the given list of encoded scenarios and compares the current solutions in the design at hand to the preference list provided by the scenario interpreter. The design is then rewarded by points from each scenario; the points are scaled so that the higher the sub-solution regarding the scenario is in the preference list, the more points are rewarded. The actual scenario fitness value is achieved by simply summing the points gained from each scenario.

Note that there are several factors affecting the solutions eventually selected to satisfy the scenarios: the degree of modifiability support for a responsibility achieved, the effect of the solution on the core fitness (including efficiency and complexity, see Subsection III.D), and the probability of the scenario. Thus, even with the very limited set of patterns we have used, the variation in the proposed architectures is remarkable.

Formally, the scenario sub-fitness function sf_s can be expressed as

$$sf_s = \sum \text{scenarioProbability} * 100/\text{scenarioPreference}.$$

Adding the scenario sub-fitness function to the core fitness function would result in the overall fitness, $f(x) = f_c(x) + w_s * sf_s$. As the scenario sub-fitness function measures modifiability, it may also be used to replace the modifiability functions sf_1 and sf_2.

V. EXPERIMENTS

We used our two sample systems, the robowar and the e-home, to see how the scenario-based fitness function affects the development of the fitness curve. In our experiments we used a population of 100 individuals and 250 generations. The curves are averages of 10 test runs. The y-value of the curve represents the average fitness of the 10 best individuals in each generation. All sub-fitness functions were given the same weight, i.e., no quality viewpoint was valued over another.

We have previously used the core fitness function to evaluate the e-home system [11, 12]. Here we are most interested in two issues: firstly, how the scenario fitness function affects the development of the fitness curves, and secondly, how the system would survive scenario evaluation if the scenario fitness was not included in the genetic algorithm.

First we measured the effect of adding the scenario fitness to the core fitness function. Figure 3 depicts the overall fitness curve for the robowar system.

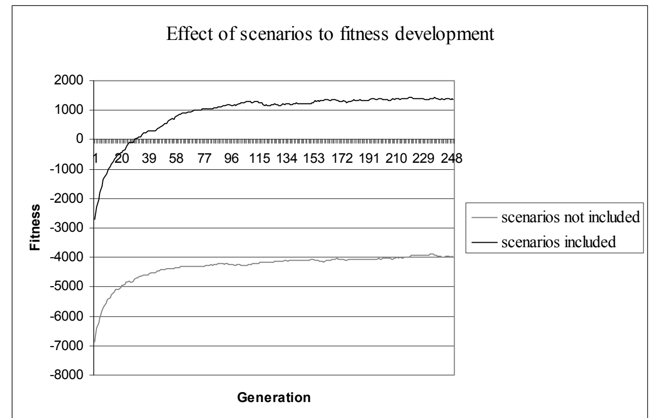


Figure 3. Altogether fitness curves for robowar

As can be seen, adding the scenario fitness naturally elevates the fitness curve on the grid as the scenario function

only gives positive values. What we are interested in is the development trend of the curves. We observe that the fitness delta (that is, the difference between the starting point and the ending point) is 4100 when scenarios are included and 3000 when they are not included. Moreover, we see that the turning points in the curve trend happen 10 to 20 generations later in the case where scenarios are included.

The same curves for the e-home system are given in Figure 4. Notice that the fitness curves start at different values, as the fitness values depend on the amount and quality of each particular system's responsibilities and their dependencies. The curves behave similarly as in the robowar graph, showing fitness delta 11000 with scenarios and 8500 without scenarios. Also in this case the first turning point in the development of the fitness curves happens approximately 20 generations later in the case where scenarios are included.

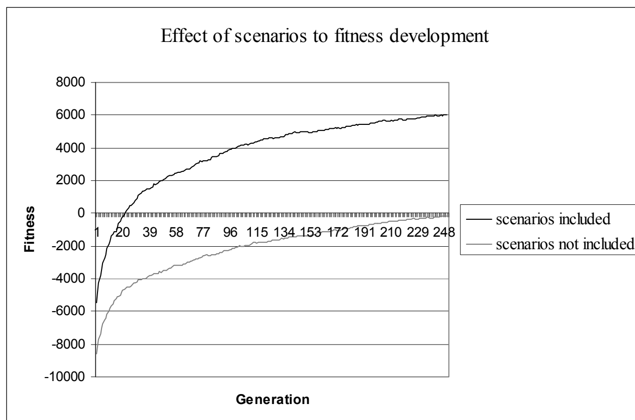


Figure 4. Altogether fitness curves for e-home

We assume that if the scenarios would only affect the curve by adding the extra positive value, the development trends would be the same, i.e., the turning points in development would be at the same point for both curves. Thus, the experiments suggest that including scenarios in the fitness function gives the genetic algorithm a slightly wider range of improvement possibilities. Although it is hard to infer the reason, we conjecture that including the evaluation of scenarios aids the design of the system in such a way that it is possible to make more good design decisions in the critical stage where the architecture is still rough enough to enable larger decisions.

The second point to investigate is how the system can handle the given scenarios when the scenario fitness value is not included in the overall fitness. This setup corresponds to a situation where the architecture, which is produced by applying the general core fitness function without scenarios, is subjected to scenario-based architecture evaluation. The curves for robowar are given in Figure 5.

It is hardly surprising that when selection is based (partly) on scenarios, the resulting architecture can handle them better. However, the growth behavior of the fitness is interesting. As can be seen, when the scenario fitness value is not included in the actual fitness function, it first begins to rise as the overall modifiability of the system increases due to the application of different patterns. However, the curve quite soon stabilizes at 5000, and no development can be seen after that.

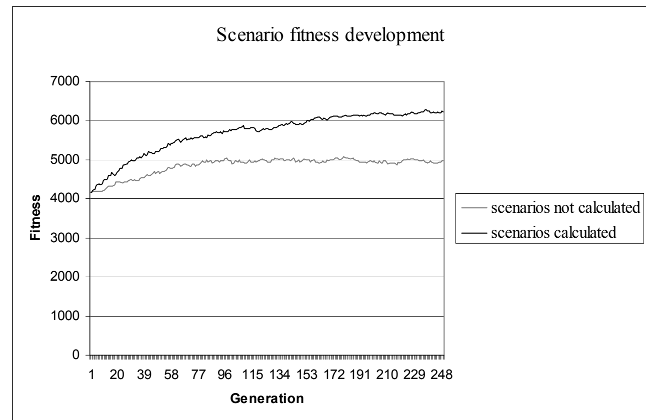


Figure 5. Scenario fitness development for robowar

The scenario fitness curve representing the case where the scenario fitness value is added to the core fitness behaves quite differently. The development over the first 50 generations is noticeably faster, and even though it somewhat slows down and stabilizes afterwards, there is still development to be seen, as the system continues to improve in terms of handling the given scenarios.

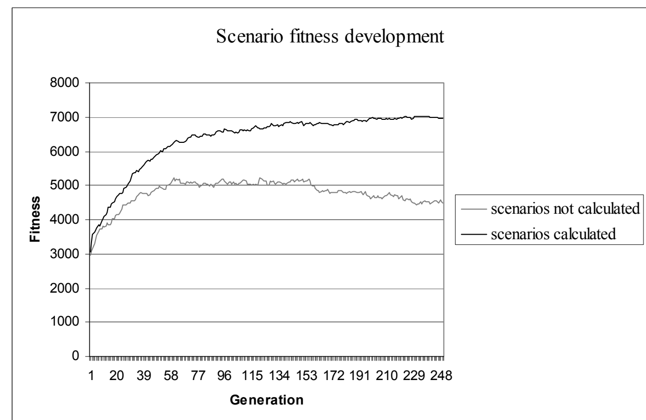


Figure 6. Scenario fitness development for e-home

Similar curves for the e-home system are given in Figure 6. In this case there is an even bigger difference in the behavior of the curves. When the scenario fitness value is not included in the overall fitness, the scenario curve first begins to rise, then quickly stabilizes, and actually starts to worsen after about 150 generations. When the scenarios are included, however, the scenario curve continues to develop rapidly until about 115 generations, after which the development slows down and the curve seems to finally stabilize to 7000. This shows that after a certain point the core fitness function would actually start to implement patterns in such a way that the architecture would handle the given scenarios very poorly. However, when the scenarios

are included in the calculations after this point, it does ensure that the architecture keeps the highest possible level it can for handling the scenarios.

Although the fitness curves give an intriguing viewpoint to how the scenario fitness affects the improvement of architecture quality we are primarily interested in the actual resulting architecture: does the use of scenarios lead to more "correct" architectures in terms of human evaluation? Although it is hard to come up with specific metrics for this purpose, we can analyze the resulting architecture proposals from this viewpoint.

Figure 7 shows a proposed architecture for the robowar system as a component diagram; this has been extracted from the actual class diagram to be more easily comprehensible. Examples of actual, whole class diagrams are given by Räihä [9].The used patterns are shown schematically as dotted boxes; an exemplary detailed UML class diagram corresponding to a fragment of the architecture is presented in Figure 9.

In the experiment resulting in Figure 7, the scenario fitness function has completely replaced the modifiability sub-functions (sf_1 and sf_2) in the core fitness function; that is, modifiability is evaluated only in terms of the scenarios. The efficiency and complexity sub-functions are, however, kept intact. The scenario sub-function is given a significantly larger weight compared to the other evaluators. This guides the use of modifiability enhancing patterns towards the responsibilities that are included in the scenarios. However, the versatility of the genetic algorithm is clearly revealed in the proposal nevertheless. For example, consider the modifiability support for Armor. The given scenario states that it is highly probable that the developer would want to add an alternative way to control the armor. Thus, a Template Method or Strategy pattern would seem a natural way of handling the situation. However, as seen in Figure 7, the GA has chosen to use Adapter instead. Since an Adapter allows the changing of the interface as well, this solution provides even stronger support for modifications than directly required by the scenarios, with fairly small cost regarding the overall complexity of the system.

Scenarios may also be supported indirectly. As can be seen in Figure 7, the findRobot responsibility has been placed behind a Strategy pattern in the Intelligence component. The related scenario states that the intelligenceControl responsibility in that component is the one that should be easily changed by the developer. Instinctively, there might be a reason to use the Template Method pattern for isolating the intelligenceControl responsibility. However, there is no method within the Intelligence component that uses the intelligenceControl responsibility, and thus the Template Method cannot be implemented here. Consequently, the GA uses an alternative way: as intelligenceControl uses two responsibilities, findRobot and controlShooting, changeability is achieved by separating findRobot behind a pattern, and thus making the intelligenceControl at least partially more modifiable.

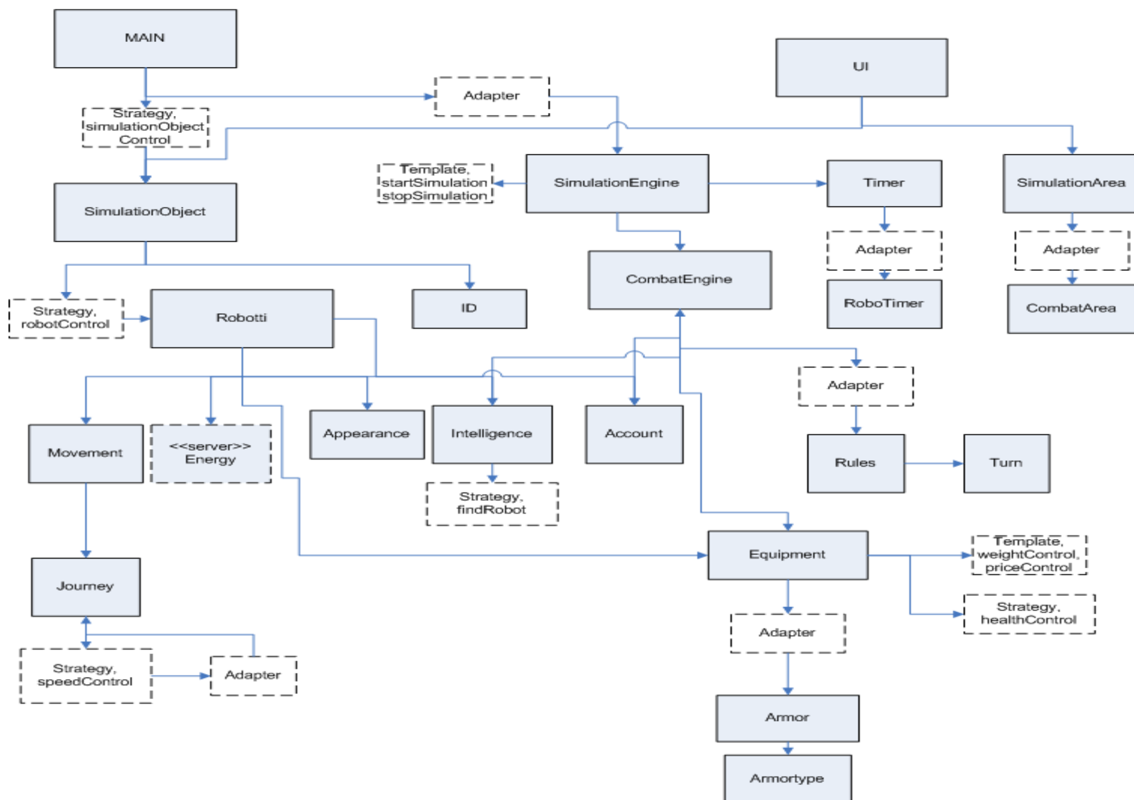


Figure 7. Example architecture for robowar system

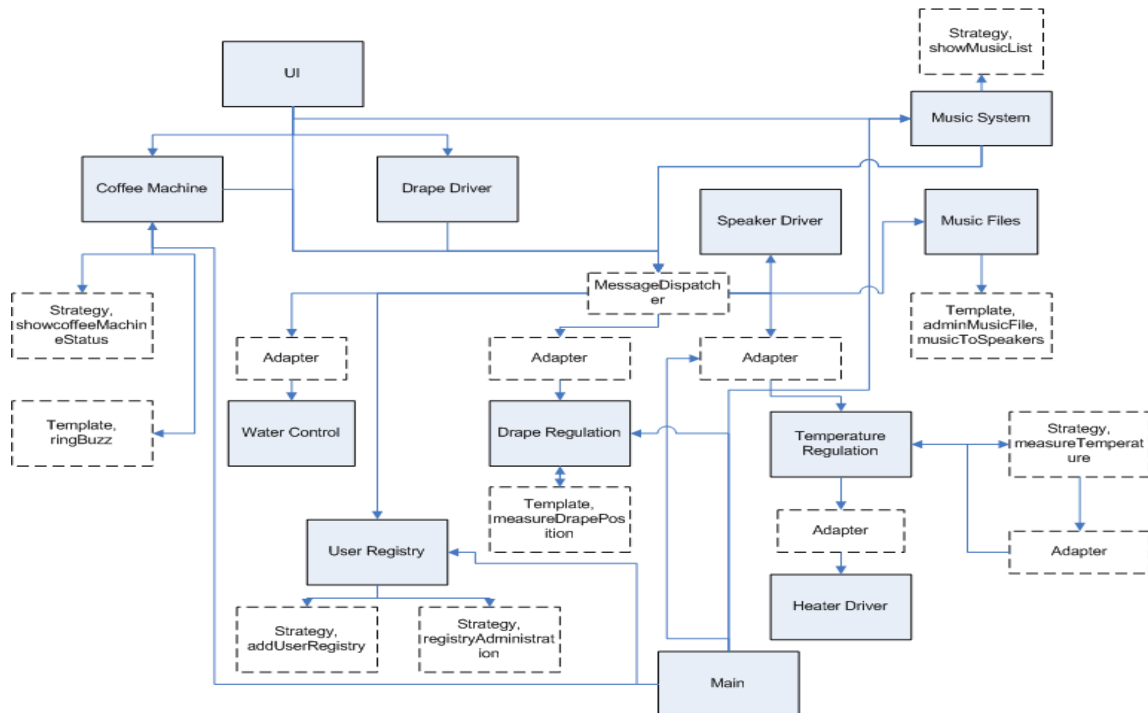


Figure 8. Example architecture for e-home

There are also direct effects of scenarios to be seen in the proposed architecture. For example, the robotControl responsibility in the Robot component is placed behind a Strategy, as an alternative implementation should be easily added. The demand to add an alternative way to control the Energy related responsibilities is solved in a curious way: the Energy component is placed behind a server. This is in a way a clever solution, as the responsibility that is tied to the scenario is in the same situation as intelligenceControl discussed above. In many solutions the situation was actually handled similarly as in the case of the Intelligence component: the responsibilities used by the "main" responsibility of a component were placed behind a Strategy or a Template instead of the "main" responsibility.

Figure 8 similarly shows an example architecture for the e-home system. In this case the general modifiability and the scenario fitness functions were heavily weighted in relation to efficiency and complexity. This kind of weighting tends to introduce the message dispatcher architecture style in the produced solution, as demonstrated by the example.

All major components except the Main Controller are involved with the message dispatcher somehow, as the message dispatcher handles the connections from UI, Coffee Machine, Music System and Drape Driver to Drape Regulation, User Registry, Temperature Regulation, Speaker Driver, Music Files and Water Control. However, the inclusion of scenario fitness can clearly be seen. Examples are the Adapter patterns controlling connections to Water Control and Heater Driver, where the operation to be called is likely to change. The Strategy and Template patterns in the Music System and the Strategy pattern in the Coffee Machine system also conform to the given scenarios, as it is highly probable that the operations behind these patterns will need to be changed or an alternative implementation should be added at some point.

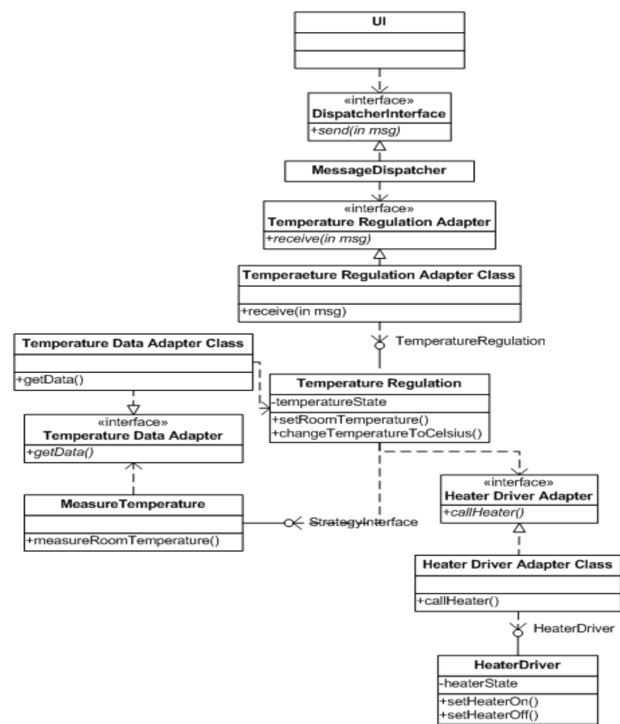


Figure 9. Example of class diagram structure for e-home

In particular, placing the showMusicList responsibility behind a Strategy is responding to the scenario given in Section IV.

In Figures 7 and 8 we have used a simplified notation for the design patterns and the architectural style to give a rough overall picture of the architectural solutions. As an example of how a design solution would look like in the standard class diagram form, Figure 9 shows the part of Figure 8 where the UI calls the Message Dispatcher, which in turn calls the Adapter that is providing a communication link to the Temperature Regulation component. Here the detailed structures of Strategy (MeasureTemperature class and its interface) and Adapter patterns are shown.

To summarize, the experiments indicate that using scenarios indeed improves the solutions produced by a genetic algorithm. This can be seen both in the general growth behavior of the fitness function and in analyzing individual architectures. Especially the architecture proposal for e-home is strikingly sensible, resembling actual human-designed architectures. Still, there is relatively much variation in the results, and the fitness weights have great influence on the character of the result. Perhaps this reflects the nature of architecture design in reality: there is no single "right" architecture, and the design is strongly affected by the relative priorities of the quality attributes.

VI. CONCLUSIONS

We have presented a method for genetically synthesizing software architectures based on functional requirements. We have discussed and shown how to use scenarios as a part of this automatic design process, and presented the data and examples achieved with this approach. Based on these results, it seems that scenarios do indeed capture the reality of architecture quality evaluation better than metrics that are commonly used when a numerical evaluation is needed.

Although we have here concentrated on modifiability scenarios, the approach is not tied to this particular quality attribute. As long as a scenario can be formalized and there is a way to measure how well the architecture supports the scenario, the scenario can be related to any quality attribute.

We acknowledge that the amount of possible patterns is limited in this work. Thus, in our future work we intend to add many more design patterns and also other architecture styles, to make the design of architecture much more flexible and closer to reality. Also,, there is still work needed in order to improve the quality of the results. For example, the balancing of different weights is not an easy task if one is aiming for a certain type of architecture. Thus, our future work includes implementing a Pareto optimal fitness function. This provides the user the option of choosing from several possible optimal solutions from the Pareto front, instead of being presented only one solution per test run, as is the case at the moment. We also intend to study the crossover of architectures in more detail. For example, choosing the parents in such a way that they complement each other's quality properties could speed up the development of the architectures considerably.

ACKNOWLEDGMENT

This work was made as part of the Darwin project, funded by the Academy of Finland.

REFERENCES

[1] M. Amoui, S. Mirarab, S. Ansari, and C. Lucas, "A genetic algorithm approach to design evolution using design pattern transformation," International Journal of Information Technology and Intelligent Computing 1, 2006, pp. 235-245.

[2] L. Bass, P. Clements and R. Kazman, Software Architecture in Practice, Addison-Wesley, 1998.

[3] S.R. Chidamber and C.F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering 20 (6), 1994, pp. 476-492.

[4] J. Clarke, J.J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating Software Engineering as a Search Problem," IEE Proceedings - Software, 150 (3), 2003, pp. 161-175.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley. 1995.

[6] M. Harman, R. Hierons and M. Proctor, "A new representation and crossover operator for search-based optimization of software modularization," In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), 2002, pp. 1351–1358.

[7] R. Kazman, M. Klein and P. Clements, ATAM: Method for architecture evaluation, Carnegie-Mellon University, Technical report CMU/SEI-2000-TR-004, August 2000

[8] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolutionary Programs. Springer-Verlag, 1992.

[9] O. Räihä, Evolutionary Software Architecture Design, University of Tampere, Department of Computer Sciences, Report D-2008-11, 2008.

[10] O. Räihä, A Survey on Search-Based Software Design, University of Tampere, Department of Computer Sciences, Report D-2009-1, 2009.

[11] O. Räihä, K. Koskimies, and E. Mäkinen, "Genetic Synthesis of Software Architecture," In: Proc. of The 7th International Conference on Simulated Evolution and Learning (SEAL'08), 2008, Springer LNCS 5361, pp. 565-574.

[12] O. Räihä, K. Koskimies, E. Mäkinen, and T. Systä, "Pattern-Based Genetic Model Refinements in MDA," In: Proc. of the Nordic Workshop on Model-Driven Engineering (NW-MoDE'08), Reykjavik, Iceland. University of Iceland, 2008, pp. 129-144.

[13] O. Räihä, E. Mäkinen and T. Poranen, Using Simulated Annealing for Producing Software Architectures. University of Tampere, Department of Computer Sciences, Report D-2009-2, 2009.

[14] M. Shaw, and D. Garlan, Software Architecture - Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[15] C.L. Simons and I.C. Parmee, "Single and multi-objective genetic operators in object-oriented conceptual software design," In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07), 2007, pp. 1957-1958.

[16] C.L. Simons and I.C. Parmee, "A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design," Engineering Optimization 39 (5) 2007, pp. 631-648.

Paper IV

Outi Räihä, Kai Koskimies and Erkki Mäkinen, Empirical Study on the Effect of Crossover in Genetic Software Architecture Synthesis, In: *Proc. of the World Congress in Nature and Biologically Inspired Computing (NaBiC'10)*, Coimbatore, India. December 2009, IEEE Press, 619-625.

© 2009 IEEE. Reprinted, with permission

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University'of Tampere's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this material, you agree to all provisions of the copyright laws protecting it.

Empirical Study on the Effect of Crossover in Genetic Software Architecture Synthesis

Outti Räihä, Kai Koskimies
Department of Software Systems
Tampere University of Technology
Tampere, Finland
outi.raiha@tut.fi, kai.koskimies@tut.fi

Erkki Mäkinen
Department of Computer Sciences
University of Tampere
Tampere, Finland
em@cs.uta.fi

Abstract — **In our previous work, we have presented a method for genetically synthesizing software architecture design. Synthesis begins with a responsibility dependency graph and domain model for a system, and results in a full architecture proposal through the application of design patterns and architectural styles. In this paper, we study the method of reproduction in the genetic algorithm. More specifically, we try to find out whether sexual or asexual method of reproduction should be used. We hypothesize that although sexual reproduction method is so favored among various species of animals and plants, asexual reproduction is more natural in the case of genetic synthesis of software architecture. We search for empirical confirmation to our hypothesis by performing tests on two sample systems.**

Keywords-software architecture; software design; genetic algorithm; search-based software engineering; sexual and asexual reproduction

I. INTRODUCTION

Design of software architecture is one of the most critical and intellectually demanding activities of software engineering. In our previous work [8, 9, 10] we have taken the viewpoint that the design of software architecture is essentially a combinatorial task where the problem is to find an optimal conformation of known good practices and solutions of architectural design in the context of a particular application. An optimal software architecture is the one which supports the realization of the quality requirements of the system under design as well as possible. Assuming that both the architecture and the general solutions can be formally represented, and that the quality of an architecture can be effectively measured, the problem of architecture design becomes a search problem that is appropriate for a heuristic search algorithm.

We have shown [8, 10] that genetic algorithms can be applied to the problem of architectural design, interpreting general solutions (such as architectural styles and design patterns) as mutations and using architectural metrics as a basis for the fitness function. The quality of an architecture was measured with respect to three quality attributes, modifiability, efficiency and understandability. The architecture objects were presented as UML class diagrams. The input of the genetic algorithm consisted of a responsibility graph, that is, a graph of tasks the application is supposed to be able to perform. This corresponds to the functional requirements of the application. In addition, a domain model of the application was given as an initial rudimentary architecture.

We have further studied in particular modifiability as a quality criterion, and shown [9] that scenario-based fitness metrics can lead to improved modifiability characteristics of the resulting architecture. Scenarios describe concrete imaginary situations that are expected to take place during the lifetime of the system. Each scenario specifies a particular modification need and its probability, and the fitness function computes the suitability of the architecture for the scenarios. In a sense, using scenarios in this way means that we imitate assumed system evolution with the simulated evolution of the genetic algorithm. However, since our preliminary tests showed that the results concerning comparison of sexual and asexual reproduction methods were almost identical in the cases of scenario-based fitness metrics and the original simpler metrics, we decided to use here the latter metrics.

In the present paper, we study the form of reproduction method used in the genetic algorithm, in particular, whether a sexual method of reproduction provides benefits when synthesizing software architectures using genetic algorithms. We consider a reproduction method in a genetic algorithm as "sexual" if it somehow combines parts of two separate individuals (parents) to form offspring. This is called cross-over. All other forms to create new individuals are considered as "asexual". In the context of genetic algorithms, they are often called mutations although in the biological sense there are several different methods of asexual reproduction [15], and the operations used might be more closely analogous to some other asexual method than natural mutation.

A sexual reproduction method can have different degrees of randomness. The parents can be chosen in a fully random way, or some characteristics of the individuals can increase their probability of pairing. When two individuals pair, the contribution of both individuals passed over to the offspring can be fully random, or it can be chosen in some sense purposefully. The latter makes sense especially if the parents have been chosen in such a way that they presumably "complete" each other (cf. breeding of domestic animals). In this work we use crossover in which selecting the contributions of each parent is fully random. However, the probability of being selected as a parent increases as an individual is ranked higher in the population. Ranking of

978-1-4244-5612-3/09/$26.00 ©2009 IEEE

individuals is based on ordering their fitness values. Similarly, the probability of being selected as a parent decreases if an individual is ranked as belonging to the worse half of the population. This corresponds best to reproduction in nature: "fit" individuals are likely to become parents, but selecting different properties is not guided. More guided ways of sexual reproductions also raise many issues that are outside the scope of this paper.

In a conventional software development process, a software architecture is recommended to be designed by making small improvements rather than by merging two architecture proposals (see e.g. the software architecture design process model in [2]). This kind of incremental design process is essentially analogous to the gradual changes typical for asexual reproduction methods. We therefore hypothesize that sexual reproduction does not offer significant advantage in the genetic synthesis of a software architecture, at least when implemented in a random fashion. The hypothesis is tested by studying the genetic synthesis of an architecture for two sample systems of different character, the control system for an e-home and a robot war game application. There seem to be only a few previous studies comparing different reproduction methods for a particular problem (see, e.g., [12, 13, 14]). In the field of search-based software engineering, it is common to either settle for a certain type of reproduction method or use only mutations without defining how natural selection is handled. Thus, our work provides experimental background that may encourage diversity and more comparative studies in reproduction methods for genetic algorithms.

The paper is organized as follows. In the next section we discuss the factors that are supposed to affect the popularity of sexual reproduction in various species and motivate our hypothesis. In Section III we describe our approach to genetically synthesize software architecture design. In Section IV we discuss our experiments and interpret their results. Finally, concluding remarks and ideas on future research directions are presented in Section V. We assume basic knowledge of genetic algorithms (e.g., Michalewicz [6]).

II. Sexual vs. asexual reproduction

Sexual reproduction is very common, especially among large, multicellular organisms, although sex is more complicated than asex. It takes more time and energy and requires selecting a partner. Moreover, recombination can both destroy and create favorable gene combinations. From a naive engineering point of view it would be much more efficient to directly move the good properties of a competent individual to the next generation without pushing one's luck with the risks of recombination. [15]

Evolutionary biologists have tried to find out reasons for the success of sexual reproduction, but no obvious general explanation has been found; most likely there are several factors promoting sexual reproduction. It is also possible that sex is maintained by different reasons in different species. However, some general explanations for explaining the popularity of sexual reproduction among species are suggested [15]:

- Recombination speeds up evolution by bringing together two or more beneficial mutations that appeared in different organisms. This is important especially when the environment changes quickly and the population must adapt to the changing circumstances.
- Recombination affects host-parasite (and predator-prey) coevolution where the direction of selection is changed relatively frequent. In a sexual population temporarily bad genotypes can retain latent and become common, when the selection again changes direction.

In our previous experiments on genetic synthesis of software architecture, we have viewed architecture as a species that evolves exploiting both mutations and crossover [8, 9, 10]. But are the biological motivations favoring sexual recombination relevant in the context of designing software architecture?

The lifetime of a software system essentially begins with the first requirements specifications and ends when the system is no more used. During this period the different artifacts representing the system experience changes that are usually referred to as the evolution of the system. Although the changes concern an individual (a particular system) rather than a species or a gene, this process bears resemblance to natural evolution. The environment where the system is developed and used changes, there are new competitors of the system on the market, the behavior of the interacting species (humans and other systems) changes etc. The main challenge of software architecture design is to somehow anticipate all (or as much as possible) such changes, and to find an optimal degree of flexibility without sacrificing other desired quality attributes (like, say, efficiency). However, there is no obvious analogy to a "population" of software architectures.

To create such an analogy, imagine that the software architecture of a system is being developed by a large team of designers following the principle of natural evolution: each individual designer proposes an initial software architecture, some authority decides which are selected for further elaboration, these are again distributed to the designers which try to improve the architectures by making small, simple modifications (mutations), and some designers are allowed to combine their solutions (crossover). The process is iterated for a certain number of times (generations), and finally the best architecture of the final round is taken as the result. This is essentially the process that has been automated in our previous experiments.

Creating a software architecture incrementally by making simple transformations is actually recommended in real software production [2], but anything that could be considered as a random crossover operation hardly ever takes place in the evolution of a system. Thus, in the fictional description above, it seems plausible that the

designers could give up combining their proposals without loss of productivity.

The research question of this paper concerns the role of crossover in genetic software architecture synthesis: does it make sense to apply (random) crossover at all in this domain, and if so, what is its effect? Based on the discussion above, our hypothesis is that sexual reproduction does not provide essential benefit for the genetic synthesis of software architecture.

III. Genetic software Architecture Synthesis

In this section we describe our approach to genetically synthesize software architecture design. We begin with a set of responsibilities (requirements) that can be given some relative values regarding modifiability and efficiency. Using the information given on the dependencies between the responsibilities, this set is then formed into a responsibility dependency graph. Furthermore, a domain model for the system is given. The graph is encoded as a chromosome, which is then subjected to the genetic algorithm (implemented with Java). The algorithm gives a detailed architecture design for the given system through the implementation of a set of architectural patterns to the given model, and produces a UML class diagram as the result.

A genetic algorithm maintains a population of possible solutions. In our problem a population is a set of possible architectures which undergoes an evolutionary process imitating the natural biological evolution. In each generation better individuals have greater possibilities to survive and reproduce, while worse individuals have greater possibilities to die and to be replaced. It is believed that this process leads to a combination of the properties of the better individuals, which constitutes a good solution to the problem in question.

To operate with a genetic algorithm, one needs an encoding of possible solutions, an initial population, mutation and crossover operators, a fitness function and a selection operator for choosing the survivors for the next generation.

A. Requirements

Any system can be described at a very high level by naming responsibilities for abstract tasks and the dependencies between them. Thus, the functional requirements for a system can be given as a responsibility dependency graph, where each node represents a responsibility, and each directed edge represents a dependency between the two responsibilities. In order to evaluate the system, some attributes of the responsibilities are also given, such as variability, parameter size and time consumption. The precise values for these attributes of course cannot be known, but should be estimated in order to calculate the modifiability and efficiency values for the system. The given values for the attributes are relative, rather than absolute.

We have used here two sample systems; an e-home control system (called hereafter ehome), which represents a typical embedded system, and a robot war game application (robowar), which represents a desktop system. The ehome system requirements lead to 52 responsibilities and 90 dependencies between the responsibilities, while robowar entails 56 responsibilities and 73 dependencies between them. A part of the ehome responsibility dependency graph is given in Figure 1. The nodes representing data manager responsibilities are marked with a thick circle.

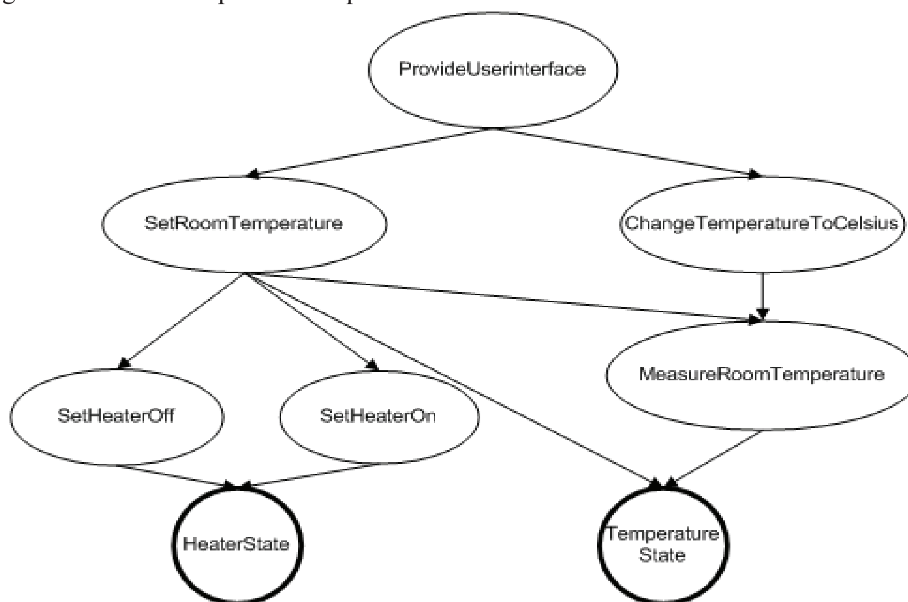


Figure 1. A fragment of the responsibility dependency graph for e-home

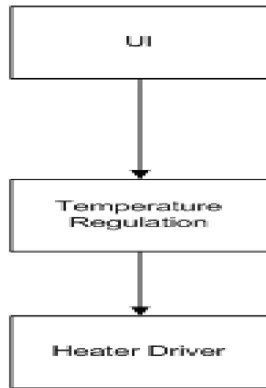


Figure 2. Fragment of domain model for ehome

In addition to simply giving the responsibilities and their relationships, we have formed a domain model by assigning the responsibilities to classes. The assignment is based on the data responsibilities: a responsibility is placed in the same class where the data it handles (either directly or through another responsibility) is. Associations for the model are derived directly from the dependency graph. This initial model thus gives the system a basic structure by separating subsystems into components. However, this structure is not fixed, as the class division may be altered through application of architectural patterns. The ehome domain model contains 12 classes and the robowar domain model contains 22 classes.

A fragment of the ehome domain model is given in Figure 2; this represents the same part of the system as Figure 1. For simplicity, we have used only classes and associations. As can be seen, the initial model is very simplistic – the genetic algorithm is merely given a logical starting point where the subsystems are separated. No decisions are made regarding the actual architectural solutions.

B. Genetic Representation

The genetic algorithm is given two kinds of data regarding each responsibility r_i. Firstly, there is the basic information of r_i itself. This contains the responsibilities $R_i = \{r_k, r_{k+1}, \ldots, r_m\}$ depending on r_i, its name, type, frequency parameter size, execution time, call cost, call variability and variability. Secondly, there is the information regarding the responsibility r_i's place in the architecture: the class(es) $C_i = \{C_{i1}, C_{i2, \ldots,} C_{iv}\}$ it belongs to, the interface it implements, the message dispatcher it uses (further explained in Section III.C), the responsibilities $RD_i \subset R_i$ that call it through the message dispatcher, the design patterns $P_i = \{P_{i1}, P_{i2, \ldots,} P_{im}\}$ it is a part of and the predetermined model class it is assigned to, as given in the domain model. The message dispatcher is given a separate field as opposed to other patterns for efficiency reasons. All data regarding a responsibility is encoded as a *supergene* [1]. The chromosome handled by the genetic algorithm is gained by collecting the supergenes, i.e., all data regarding all responsibilities, thus achieving a whole view of the functional requirements for the architecture.

The initial population is made by first creating the desired number of individuals with the basic structure given in the responsibility dependency graph and domain model. A random pattern is then inserted into each individual, as a population should not consist entirely of clones. In addition, a special individual is left in the population where no pattern is initially inserted.

C. Mutation and Crossover Operations

The actual architectural design means here the application of various standard architectural solutions called collectively patterns. The patterns have been chosen to represent solutions on different levels: high-level architectural styles [11] (message dispatcher and client-server), medium-level design patterns [4] (Façade and Mediator), and low-level design patterns [4] (Strategy, Adapter and Template Method). The mutations are implemented in pairs of introducing a pattern or removing a pattern [7]. The dispatcher architecture style makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the responsibilities can communicate through it. Although the number of patterns is very small in our experiments, we assume that this is sufficient for studying the basic characteristics of genetic architecture synthesis.

The mutations are thus the following:

- introduce/remove message dispatcher
- create link/remove link to dispatcher
- introduce/remove server
- introduce/remove façade
- introduce/remove mediator
- introduce/remove strategy
- introduce/remove adapter
- introduce/remove template method.

The crossover operation is implemented as a traditional one-point crossover with a corrective operation. The latter operation ensures that the architecture stays coherent, as patterns may be broken by overlapping mutations. In addition to ensuring that the patterns present in the system stay coherent and "legal", the corrective function also checks that the design conforms to certain architectural laws. These laws demand uniform calls between two classes (interaction either through an interface or a dispatcher but not both), and state some basic rules regarding architectures (e.g. no responsibility can implement more than one interface). The purpose of these laws is to ensure that no anomalies are brought to the design.

The actual mutation probabilities are given as input. Selecting the mutation is made with a "roulette wheel" selection [6], where the size of each slice of the wheel is in proportion to the given probability of the respective mutation. Null mutation and crossover (if applied) are also included in the wheel. Each individual has a chance of reproducing in each generation: if the first roulette selection lands on a mutation, another selection is performed after the mutation has been administered. If the second selection lands on the crossover slice, the individual may produce offspring. In any other case, the second selection is not taken into account, i.e., the individual is not mutated twice.

D. Fitness Function

The fitness function is based on widely used software product metrics, most of which are from the metrics suite introduced by Chidamber and Kemerer [3].These metrics, especially coupling and cohesion, have been used as a starting point for the fitness function, and have been further developed and grouped to achieve clear "sub-fitnesses" for modifiability and efficiency, both of which are measured with a positive and negative sub-function. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-sever architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. A complexity metric is added to penalize having many classes and interfaces as well as extremely large classes.

Dividing the fitness function into sub-functions answers the demands of the real world. Hardly any architecture can be optimized from all quality viewpoints, but some viewpoints are ranked higher than others, depending on the demands regarding the specific architecture at hand. By separating efficiency and modifiability, which are especially difficult to optimize simultaneously, we can assign a bigger weight to the more desired quality aspect. When w_i is the weight for the respective sub-fitness sf_i, the fitness function $f_c(x)$ for chromosome x can be expressed as

$$f_c(x) = w_1 * sf_1 - w_2 * sf_2 + w_3 * sf_3 - w_4 * sf_4 - w_5 * sf_5.$$

Here, sf_1 measures positive modifiability, sf_2 negative modifiability, sf_3 positive efficiency, sf_4 negative efficiency and sf_5 complexity. The sub-fitness functions are defined as follows (|X| denotes the cardinality of X):

sf_1 = |interface implementors| + |calls to interfaces| + |calls through dispatcher| ∗ ∑ (variabilities of responsibilities called through dispatcher) – |unused responsibilities in interfaces| ∗ α,

sf_2 = |calls between responsibilities in different classes, that do not happen through a pattern|,

sf_3 = ∑ (|dependingResponsibilities within same class| ∗ parameterSize) + ∑(|usedResponsibilities in same class| ∗ parameterSize + |dependingResponsibilities in same class| ∗ parameterSize),

sf_4 = ∑ ClassInstabilities + (|dispatcherCalls| + |serverCalls|) ∗ ∑ callCosts, and

sf_5 = |classes| + |interfaces|.

The multiplier α in sf_1 emphasizes that having unused responsibilities in an interface is almost an architecture law, and should be more heavily penalized.

E. Selection

In sexual reproduction selection is natural, as crossover adds individuals to the population while the size of the population should be the same in the beginning of each generation. Thus, a selection method is needed to discard the required amount of individuals. For this purpose, we have used the same kind of roulette wheel method as was used for choosing the mutations. This is combined with elitism to ensure that the very top of each population is kept for the next generation.

For asexual reproduction, the question of implementing selection in the genetic algorithm is not as straightforward. In order to have more variety in the population and to be able to have an option to discard the weakest individuals, natural selection should be simulated. Because crossover is not an option as a means to increase the size of the population, we have chosen to clone each individual before it is mutated. Thus, after the selection has been made after each generation, the initial population for a given generation is copied so that we have three copies of each individual, and these copies are mutated. The selection operator (elitism combined with roulette wheel) now has a vast pool of material from which to choose the next generation.

IV. Experiments

We used two sample systems, the robowar and the ehome to see how the different reproduction methods affect the development of the fitness curve. In our experiments we used a population of 100 individuals and 500 generations. The curves are averages of 10 test runs. The y-value of the curve represents the average fitness of the 10 best individuals in each generation. We are interested in two issues: firstly, is the use of (random) crossover justified, i.e., does sexual reproduction provide any added value in this domain, and secondly, what is the effect of crossover when compared to a purely asexual approach.

The fitness curves for tests on the ehome system are depicted in Figure 3. As can be seen, in the case of asexual reproduction (i.e., no crossover), the fitness curves develop very rapidly, and then stabilize and get stuck to a value just above 2000. In the case of sexual reproduction (i.e., with crossover), the fitness values continue to develop for a significantly longer time before stabilizing. However, the actual fitness value reached is noticeably lower than the value reached by asexual reproduction.

The fitness curves for tests on the robowar system are given in Figure 4. The results are very similar to the ehome system regarding the effect of crossover. Given the rather different character of the systems, these results suggest that in genetic software architecture synthesis, where patterns serve as mutations, random crossover does not provide benefits in terms of reaching high fitness values more efficiently. In contrast, pure mutation based evolution appears to lead to more rapid convergence to a stable fitness value, even on a higher level than in the case of crossover.

The difference between the shapes of the curves is not as drastic with robowar as with ehome. This can be explained by the different character of the starting points: for robowar, the classes in the initial domain model contained 2.5 responsibilities in the average, while in ehome this figure was 4.3. This implies that the initial architecture was essentially more fine-grained in robowar, leaving more possibilities for crossover to yield descendants.

In both systems the fitness curve for the asexual case appears to be after it reaches a certain level. We continued to investigate this by computing the average deviation within the last population. The average deviations of the last populations of 10 test runs are given in Table 1.

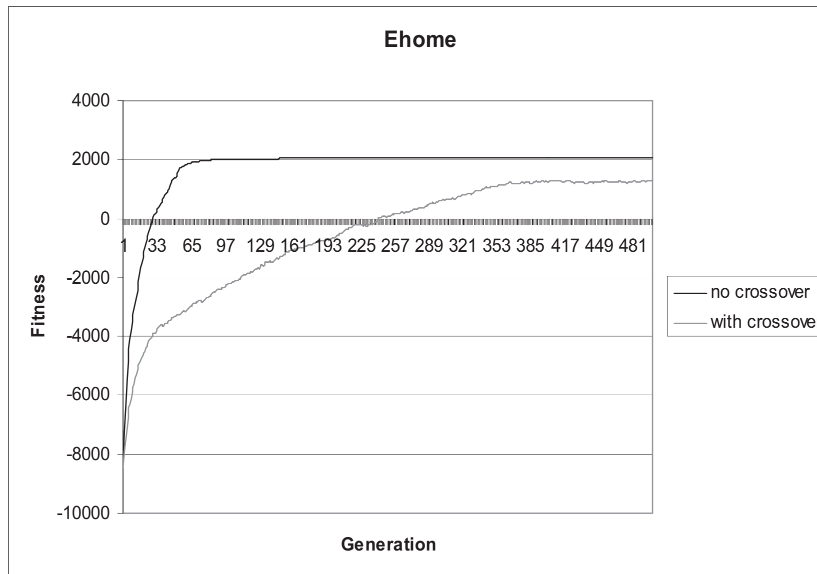


Figure 3. Ehome fitness curves

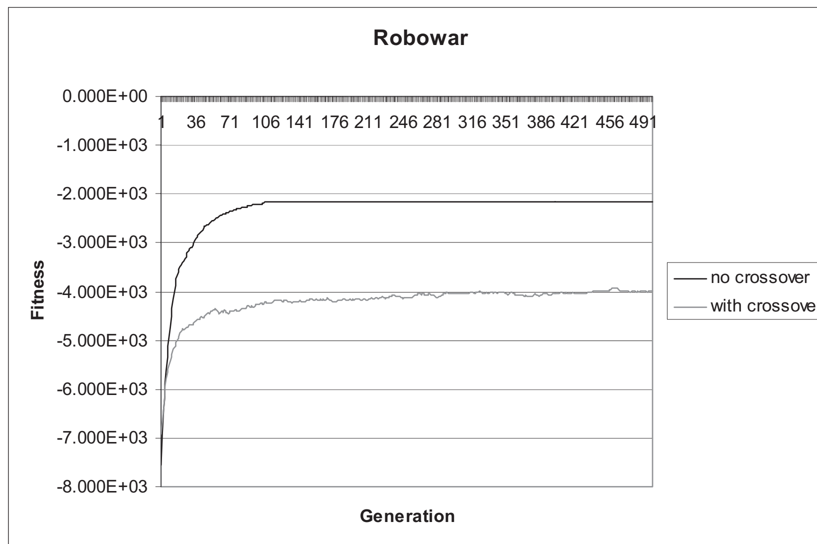


Figure 4. Robowar fitness curves

TABLE I. AVERAGE DEVIATIONS IN THE LAST POPULATION

	Ehome	Robowar
Whole population	8530.4	4130.1
Top 200	0.0164	4.77E-12

These results show that when the entire population, i.e., all 300 individuals (after mutation but before selection), is taken into account, there is still a significant amount of deviation as there are some noticeably bad individuals. However, when considering the top 200 individuals (which the next generation of 100 individuals will extremely likely consist of), there is hardly any deviation. In fact, in four test runs for the robowar system, the deviation for the top 200 individuals was zero.

These results suggest that the populations achieved with purely asexual reproduction end up consisting of individuals with identical or nearly identical fitness values. An explanation could be that the population consists of clones of some top individual, and the algorithm is not able to find any mutations that would further improve the found solution. By comparison, average deviations in the last population (after mutation and crossover but before selection) when using sexual reproduction were around 40 000 for ehome and 20 000 for robowar, when the largest population was 110 individuals.

A general observation regarding the architectures produced by purely asexual reproduction was that the selection of applied patterns tends to be reduced. In both systems, most of the best individuals of the last generations applied only Template Method and Adapter, which are low-level patterns with specific immediate positive impact on modifiability and minor negative impact on efficiency and complexity. With crossover, the selection of applied patterns was significantly larger. However, the selection of the patterns and the effect of crossover are dependent on the relative weights of the quality attributes. If modifiability is weighted over efficiency and simplicity, high-level architectural style patterns (message dispatcher and client-server) start to appear more in the best architectures, and in those cases crossover actually results in better fitness values, in contrast to the situation where several quality attributes are equally weighted. The high-level patterns differ from low-level patterns in that they are more significant for the fitness: they have rather strong positive effect on modifiability but also a clear negative effect on efficiency. Obviously, if modifiability is sufficiently emphasized, such patterns will prevail. However, the reason why crossover seems to provide advantage when the fitness function is targeted to a single quality attribute with strongly supporting patterns remains unclear. We can only speculate that crossover makes it easier to pass especially good solutions to a large number of descendants.

V. CONCLUSIONS

We have presented experimental results on the effect of sexual reproduction in the case of genetically synthesizing software architecture. We hypothesized that, considering the conventional design process of software architecture, crossover does not provide significant benefits, as there is no equivalent for this operation in the real design process. The experimental results we have presented seem to confirm this hypothesis, as the fitness curves for two sample systems achieved higher and more convergent values when using only asexual reproduction. The experiments revealed that both the individuals and the populations tend to become homogeneous with asexual reproduction, the former in the sense of used patterns and the latter in the sense that the elite consists of identical or almost identical individuals.

However, it would be too hasty to conclude that asexual reproduction should be favored in genetic software architecture synthesis. First, the criteria for good software architecture, as represented by the fitness function based on general software metrics, is highly controversial. Indeed, our previous experiments with crossover have produced architectures that are closer to human-designed ones than those produced without crossover in this work. It seems that crossover produces larger diversity in the architectural solutions, including those that "look" good for the human architect. The problem is how to identify these mechanically in the fitness function. Also, our additional experiments in this work indicated that the effect of crossover depends on the weighting of the quality attributes in the fitness function.

Second, we have used here only random crossover. While random crossover is indeed unnatural in the context of software architectures, purposeful crossover is not. It is not uncommon in practice that two persons in a design team come up with good solutions for different parts of the systems, and merge their solutions. Thus, in our future work, we intend to implement a crossover operator that would search for and then attempt to combine quality "sub-architectures" of the system. Building block conserving crossover operators has already been studied by, e.g., Harman et al. [5].

ACKNOWLEDGMENT

This work has been funded by the Academy of Finland (project Darwin).

REFERENCES

[1] M. Amoui, S. Mirarab, S. Ansari, and C. Lucas, A genetic algorithm approach to design evolution using design pattern transformation. *International Journal of Information Technology and Intelligent Computing* **1**, 2006, pp. 235-245.

[2] J. Bosch, , *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*, ACM Press, 2000.

[3] S.R. Chidamber and C.F. Kemerer, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* **20** (6), 1994, pp. 476-492.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.

[5] M. Harman, R. Hierons and M. Proctor, A new representation and crossover operator for search-based optimization of software modularization. In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*,2002, 1351-1358.

[6] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolutionary Programs.*. Springer-Verlag, 1992.

[7] O. Räihä, Evolutionary Software Architecture Design. University of Tampere, Department of Computer Sciences, Report D-2008-11, 2008.

[8] O. Räihä, K. Koskimies, and E. Mäkinen, Genetic synthesis of software architecture. In: *Proceedings of The 7th International Conference on Simulated Evolution and Learning (SEAL'08)*, 2008, Springer LNCS 5361, pp. 565-574.

[9] O. Räihä, K. Koskimies, and E. Mäkinen, Scenario-based genetic synthesis of software architecture. To appear in *Proc. of the Fourth International Conference on Software Engineering Advances 2009*.

[10] O. Räihä, K. Koskimies, E. Mäkinen, and T. Systä, Pattern-based genetic model refinements in MDA. To appear in *Nordic Journal of Computing*, 2009.

[11] M. Shaw, and D. Garlan, *Software Architecture - Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[12] A. Simões, and E. Costa, Transposition versus crossover: an empirical study, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, 1999, 612-619.

[13] A. Simões, and E. Costa, Using genetic algorithms with sexual or asexual transposition: a comparative study, In: *Proceedings of the 2000 Congress on Evoluationary Computation (CEC'2000)*, 2000, 1196-1203.

[14] A. Simões, and E. Costa, On biologically inspired genetic operators: transformation in the standard genetic algorithm, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'01)*, 2001, 584-591.

[15] S. C. Stearns, and R.F. Hoekstra, *Evolution: An Introduction*. Oxford University Press, 2000.

Paper V

Hadaytullah, Sriharsha Vathsavayi, Outi Räihä and Kai Koskimies, Tool Support for Software Architecture Design with Genetic Algorithms, In: *Proc. of the 5th International Conference on Software Engineering Advances (ICSEA'10)*, Nice, France. August 2010, IEEE CS Press, 359-366.

© 2010 IEEE. Reprinted, with permission.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University'of Tampere's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this material, you agree to all provisions of the copyright laws protecting it.

Tool Support for Software Architecture Design with Genetic Algorithms

Hadaytullah, Sriharsha Vathsavayi
Department of Software Systems
Tampere University of Technology
Tampere, Finland
{firstname.lastname@tut.fi}

Outi Räihä, Kai Koskimies
Department of Software Systems
Tampere University of Technology
Tampere, Finland
{firstname.lastname@tut.fi}

Abstract— **Automated support for software architecture design is discussed. The proposed approach is based on a tool applying genetic algorithms for producing potential architecture proposals. The tool requires a basic functional decomposition of the system and the specification of the quality requirements as input, relying on a repository of standard solutions like patterns and architectural styles. The underlying techniques and the design of the tool are discussed, and the usage of the tool is illustrated by an example.**

Keywords-software architecture; heuristic methods; genetic algorithms; patterns; tool support.

I. INTRODUCTION

Software architecture design has been traditionally considered as highly creative work, requiring special experience, judgment and talent. On the other hand, there is a growing body of architectural knowledge [2], expressed in terms of architectural styles, patterns, best practices, reference architectures etc. In many cases, a good software architecture is obtained essentially by applying existing architectural knowledge in the context of a particular system, rather than creating completely new solutions. Assuming that software architecture can be presented and evaluated in a formal manner, the design of software architecture can be thus viewed as a search problem: find a configuration of existing solutions that optimizes the architecture with respect to certain quality requirements in the context of a particular system.

This observation gives rise to our basic research question: could it be possible to automate software architecture design, at least to some extent? We argue that (partial) automation of software architecture design is beneficial not only for increasing the productivity of a software architect, but also for improving the quality of the design. The latter expectation is based on the fact that an automated design may have access to and consider without prejudice a much larger solution and knowledge base than a human software architect who often suffers from the Golden Hammer syndrome [4]: once a person has found a solution that works nicely in one context, he or she tends to apply it over and over again, even in inappropriate contexts.

We adopt the viewpoint that software architecture is the composition of a set of architectural decisions [12]. In our context, an architectural decision is a structural solution imposed on the target system, so as to satisfy the quality requirements of the system. Such solutions are typically architectural styles [24] or design patterns [10].

Heuristic search methods have been used in various fields of software engineering [7], also in areas close to software architecture [1, 13]. In our previous work, we have studied the application of genetic algorithms (GA) to software architecture synthesis [21, 22, 23]. The results suggest that it is possible to produce a reasonable software architecture using this approach, at least for domains that are well understood.

In this paper we study more pragmatic aspects of automated software architecture generation: what is the design process in this approach, and what kind of tool support could be useful for a software architect in this process? The main contributions of this paper are a proposal for automated software architecture design process and a prototype tool for automated software architecture synthesis, based on the GA approach developed in our earlier work. The empirical evaluation of the approach has been presented elsewhere [23] and is outside the scope of this paper.

We proceed as follows. In Section II we discuss the general model of (partially) automated software architecture design. In Section III we briefly discuss the related work, using meta-heuristic search methods in software design. Our approach for genetic software architecture synthesis is summarized in Section IV. The prototype tool developed in this work is presented in Section V, and an example of the usage of the tool is discussed in Section VI, illustrating the concept of automated software architecture design proposed in this paper. The performance of the tool is briefly discussed in Section VII. Finally, the paper is concluded in Section VIII.

II. AUTOMATED SOFTWARE ARCHITECTURE DESIGN

The overall process for automated software architecture design is depicted in Fig. 1. There are two major phases in the process: (i) the tool-assisted generation of a draft architecture based on a null architecture, system requirements, and a solution base; and (ii) the (manual) completion of the draft architecture, using tacit expert knowledge of the software architect. We expect that it is unrealistic to aim at fully automated software architecture design, but we argue that a useful first approximation for the software architecture can be produced by a tool.

978-0-7695-4144-0/10 $26.00 © 2010 IEEE
DOI 10.1109/ICSEA.2010.61

IEEE computer society

The tool producing a draft proposal can be based on various meta-heuristic search approaches, like simulated annealing [11], hill-climbing [11], or (as in our case) genetic algorithms (GAs) [16]. However, since there are no known methods to produce deterministically an optimal software architecture for given requirements, the tool is expected to produce a (small) set of candidate proposals which are subject to human selection. In this way the inherent random element typical for heuristic methods can be largely effaced: a good architecture that just happens to score less than another generated architecture will be then considered, too.

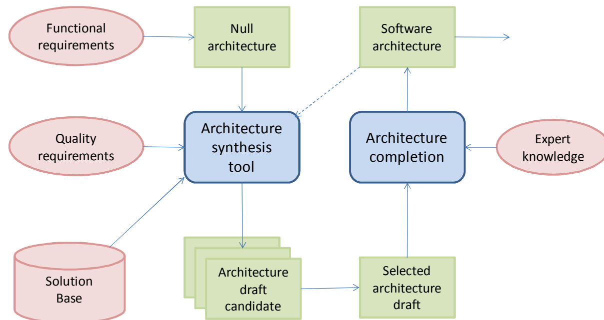


Figure 1. Overview of automated software architecture design process based on heuristic search

For a given target system, the central input consists of a *null architecture* and the system requirements. The null architecture is needed for a starting point for architecture synthesis: it gives a basic initial functional decomposition of the system in terms of major components and their responsibilities. As such, it is a rudimentary architecture that does not yet take any quality requirements into account. The null architecture is the "seed" of the tool, a structure into which various architectural solutions can be attached.

System requirements consist of functional requirements and quality requirements. Any architecture produced by the tool is assumed to satisfy the functional requirements, with varying degree of quality. The quality requirements must be given in a way that allows the algorithmic evaluation of the quality of an individual architecture. In Section IV we will discuss a possible technique to give such requirements.

We assume the existence of a knowledge base of architectural solutions. Here we do not give any requirements for the nature or presentation of those solutions - they can be general pattern-like solutions or architectural styles [3], or more specific solutions related to a particular domain. Each solution also specifies in some way the preconditions for applying the solution. In Section IV we will discuss in more detail the solution base we have used in our GA based approach.

In addition to a null architecture, the tool should also be able to take a real architecture as input. This is useful in many ways. Tool-assisted software architecture design can be made iterative: after manual improvement of the generated architecture, the architect can again submit the completed architecture to the tool, possibly freezing some parts of the architecture. The tool can use this as the null architecture, and further increase the quality by adding new solutions. For example, the architect could add manually a preferred solution for a particular problem, and freeze that so that it will be retained by the tool which tries to find optimal architectures in the presence of the frozen parts. On the other hand, the architect can also apply the tool only after making already some design decisions, which will be frozen. In this way the architect can apply the tool at any stage of the design, whenever appropriate. This is shown by the dashed arrow in Fig. 1. However, this kind of iterative design is not yet supported in our tool, discussed in Section V. In the following chapters we propose an approach for developing an architecture synthesis tool based on GA.

III. Related Work

There have been several studies describing tools using meta-heuristic search algorithms for some part of software design/re-design process, as well as tools for software architecture design. The common denominator for these tools is that they improve the software architecture rather than create it based on the requirements of the system. Also, the improvements are mostly limited to class hierarchy and decomposition, while our approach takes into account more refined compounds (patterns and architectural styles).

O'Keeffe and Ó Cinnéide [18] present Dearthóir, a tool for improving a design with respect to a conflicting set of goals by using simulated annealing. It restructures a class hierarchy and moves methods within it in order to minimize method rejection, eliminate code duplication and ensure super classes are abstract when appropriate. All refactorings are reversible, behavior-preserving transformations in Java code. A set of metrics is used for evaluating the design quality.

O'Keeffe and Ó Cinnéide [19, 20] have continued their research by constructing CODe-Imp, a tool for refactoring object-oriented programs to conform more closely to a given design quality model. It can be configured to operate using various subsets of its available automated refactorings, various search techniques, and various evaluation functions based on combinations of established metrics.

Mancoridis et al. [14] present the Bunch tool for automatic modularization. It uses hill climbing and GA to aid its clustering algorithms. A hierarchical view of the system organization is created based solely on the components and relationships that exist in the source code. The goal of this software modularization process is to automatically partition the components of a system into clusters (subsystems) so that the resulting organization concurrently minimizes inter-connectivity while maximizing intra-connectivity.

Mitchell et al. [16] build on the Bunch tool in their two step process for reverse engineering the software architecture of a system directly from its source code. Bunch is used for the first step: clustering the modules from the source code into subsystems. The second step involves reverse engineering the subsystem-level relations using a formal (and visual) architectural constraint language. Using the reverse engineered subsystem hierarchy as input, a second tool, ARIS, is presented to enable software developers to specify the rules and relations that govern how modules and

subsystems can relate to each other. ARIS then attempts to find the missing style relations.

Di Penta et al. [9] build on these results and present a software renovation framework (SRF), a toolkit that covers several aspects of software renovation, such as removing unused objects and code clones, and refactoring existing libraries into smaller ones. Refactoring has been implemented in the SRF using a hybrid approach based on hierarchical clustering, GAs and hill climbing, also taking into account the developer's feedback. Most of the SRF activities deal with analyzing dependencies among software artifacts.

The tool that is closest to ours with respect to the goal is ArchE, developed at SEI [15, 25], which, however, do not use a meta-heuristic search algorithm but a deterministic approach together with user interaction.

ArchE uses three different types of input: quality attribute requirements, the set of features (functions) that the system should support, and legacy design, if available. From the features and quality attribute requirements, ArchE constructs a representation of the responsibilities and the dependencies among them. The architect interacts with ArchE to further identify the dependencies among the responsibilities and to provide properties that are required in order to predict quality attribute behavior.

ArchE then creates an initial architecture and shows it to the architect, as well as a series of suggestions for improvements. The architect selects an option, and ArchE applies it to the architecture, calculates the effects of the revision, and shows the revised information. The interaction/revision continues until the architect is satisfied with the design [25].

IV. GENETIC SOFTWARE ARCHITECTURE SYNTHESIS

In this section we describe in more detail our basic approach to synthesize software architecture design [21, 22, 23] based on genetic algorithms. Genetic algorithms [16] are generally used to find a good solution from a very large search space, the goal obviously being that the found solution is as good as possible. Each solution is encoded as a *chromosome*, which can be further divided into *genes*. When reproducing, *crossover* occurs: genes are exchanged between the pair of parent chromosomes. The offspring is subject to *mutation*, where gene values are changed. The *fitness* represents the quality of a solution. The set of chromosomes at hand at a given time is called a *population*.

A. Functional Requirements

For any system, the functionality can be described with use cases which are refined into interactions between the major units of the system. In this way, basic responsibilities of the units can be inferred, as well as the mutual dependencies of the responsibilities (a responsibility may rely on another responsibility). In actual implementation, the responsibilities typically become the services of the units. In this work, we assume that functional requirements are given as a set of use cases which is then refined into a responsibility dependency graph (RDG), where each node represents a responsibility, and each directed edge represents a dependency between the two responsibilities (that is, the source responsibility requires the target responsibility).

In order to facilitate the evaluation of the architecture of the system, the responsibilities can be associated with qualifying attributes, such as sensitiveness to variation during the evolution of the system and average time consumption. The values for these attributes of course cannot be known precisely for a system under design, but estimations of these (relative) values provide more information for the genetic algorithm, regarding modifiability and efficiency fitness.

In our technique, the null architecture (see Fig. 1) is derived from the use cases as well: the major units obtained in the refinement of the use cases become components in the null architecture. The dependencies between the responsibilities imply directly dependencies between the corresponding components. The null architecture can be thus given as a class diagram in UML.

In order for the genetic algorithm to operate with the architectural data, it is encoded into a chromosome form. In the chromosome, each responsibility is encoded into one gene, while each data component (qualifying attributes, dependencies to other responsibilities, and its structural "place" in the architecture) is given a separate field within the gene.

The initial population is made by first creating the desired number of individuals with the basic structure given in the null architecture. A random pattern is then inserted (in a randomly chosen place) into each individual, as an initial population should not consist entirely of clones. In addition, a special individual is left in the population where no pattern is initially inserted: this ensures versatility in the population.

B. Mutation and Crossover Operations

As discussed above, the actual architectural design consists of the applications of standard architectural solutions called collectively *patterns* here. The patterns have been chosen so that there are representatives of very high-level architectural styles [24] (dispatcher and client-server), medium-level design patterns [10] (Façade and Mediator), and low-level design patterns [10] (Strategy, Adapter and Template Method). Also, the notion of interface is considered an architectural solution. The mutations are implemented in pairs of introducing a pattern or removing a pattern. The dispatcher architecture style [24] makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the responsibilities can communicate through it. The mutations are thus the following:

- introduce/remove message dispatcher
- create link/remove link to dispatcher
- introduce/remove server
- introduce/remove façade
- introduce/remove mediator
- introduce/remove strategy
- introduce/remove adapter
- introduce/remove template method
- introduce/remove interface.

The crossover operation is implemented as a traditional one-point crossover with a corrective operation. This operation ensures that the architecture stays coherent, as patterns may be broken by overlapping mutations. In addition to ensuring that the patterns present in the system stay coherent and "legal", the corrective function also checks that the design conforms to certain architectural laws. These laws demand uniform calls between two classes (e.g., through an interface or a dispatcher), and state some basic rules regarding architectures, e.g., no responsibility can implement more than one interface. The purpose of these laws is to ensure that no anomalies are brought to the design.

The actual mutation probabilities are given as input. Selecting the mutation is made with a "roulette wheel" selection [16]. Null mutation and crossover are also included in the "wheel". Each individual has a chance of reproducing in each generation: if the first roulette selection lands on a mutation, another selection is performed after the mutation has been administered. If the second selection lands on the crossover slice, the individual may produce offspring. In any other case, the second selection is not taken into account, i.e., the individual is not mutated twice.

C. *Fitness Function and Selection*

In order for the genetic algorithm to know what kinds of individuals are "good", a fitness function is needed. In our approach the fitness function, or the weights of its different parts, represent the quality requirements. In addition, more specific modifiability requirements may be given as so-called scenarios, discussed below.

The core fitness function is based on widely used software product metrics, most of which are from the metrics suite introduced by Chidamber and Kemerer [6]. These metrics, especially coupling and cohesion, have been used as a starting point for the core fitness function, and have been further developed and grouped to achieve clear "sub-fitnesses" for modifiability and efficiency, both of which are measured with a positive and negative sub-function. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-sever architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. A complexity metric is added to penalize having many classes and interfaces as well as extremely large classes.

Dividing the core fitness function into sub-functions answers the demands of the real world. Hardly any architecture can be optimized from all quality viewpoints, but some viewpoints are ranked higher than others, depending on the demands regarding the specific architecture at hand. By separating efficiency and modifiability, which are especially difficult to optimize simultaneously, we can assign a bigger weight to the more desired quality aspect. A detailed description of the basic fitness function can be found in [23].

In addition to these metric-based sub-fitness functions, we have also included modifiability related scenarios as a way of measuring the architecture quality [23]. Each scenario is also associated with a probability value, indicating the likelihood of the scenario and thus its relative weight in the evaluation. The scenarios are encoded in such a way that they can be taken into account in the calculation of the fitness value. The fitness function is extended with a term evaluating the appropriateness of the architecture with respect to the scenarios.

Selecting the individuals for each generation is made with the same kind of roulette wheel method as was used for choosing the mutations. A selection operation is needed, as the size of the population should be the same at the start of each generation, but through crossover the amount of individuals grows.

V. Tool Architecture

The prototype tool named Darwin was implemented as an Eclipse's plugin as shown in Fig. 2. Eclipse plugin architecture provides us with the facility to introduce new features into the existing workbench [26]. Darwin is based on the Genetic Algorithm (GA) Engine plugin. The GA Engine plugin contains the genetic algorithm from [23], which synthesizes the architectures. The algorithm has been slightly modified to work along with Darwin.

To implement the methodology mentioned in Section II, it was essential to integrate Darwin with a CASE tool. The UML [28] diagram editors of the CASE tool were needed to view and modify the architectures (e.g. null architecture and generated architectures) and RDG. Moreover, the editors were also required to draw other related diagrams (e.g. use case diagrams). Therefore UML2Tools plugin, which is an Eclipse based CASE tool, was incorporated for this purpose [29].

Furthermore, to understand the origins of different imperfections in a generated architecture, a graph (later named a family tree) presenting the history of the architecture was desired. To draw such a graph, we have used the Eclipse's visualization toolkit called Zest [27]. Finally, Darwin makes use of JFreeChart plugin [30] to plot the fitness vs. generation graphs. The above mentioned features will be further elaborated in the following section with examples.

The main underlying architectural style is the Model View Controller [5], as shown in Fig. 3. The model is the Evolution, which is a container for all the information regarding the generations of architectures, including the user provided inputs as well as the outputs from the genetic algorithm. The inputs consist of the RDG, the null architecture, periods with their parameters (explained later), and scenarios. The outputs consist of the generated architectures and their fitness values.

We have introduced the notion of a *period* to accommodate for the environmental changes during evolution: the environment does not necessarily remain the same throughout the evolution. We have found this useful in architectural genetic synthesis, allowing different genetic parameters for different generation ranges. For example, one could change the available pattern mutations (or their probabilities) in such a way that more fundamental patterns (like architectural styles) are introduced in the beginning, while more detailed patterns become available later. This implies that detailed

patterns are attached to the style solutions, rather than the other way around. Basically, a period is a range of generations with the same set of genetic parameters.

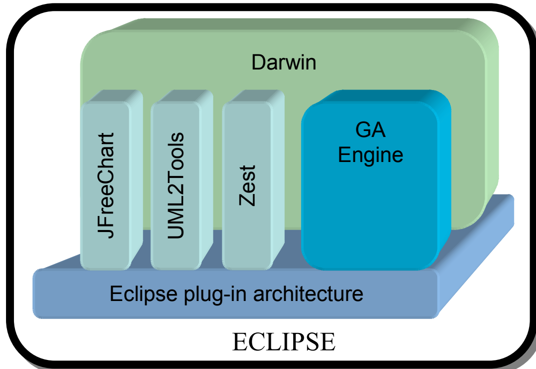


Figure 2. Darwin plugin

The genetic parameters include the mutation probabilities, generation settings and fitness weights. The mutation probabilities are the probabilities of different mutations involved in the synthesizing algorithm. The settings comprise preferences for the number of generations in the period, population size and the fitness calculation method. There are two options available to calculate the fitness of a generation in a period. It can be calculated either by averaging the elite fitnesses or otherwise just considering the fitness of the best architecture in the generation. The weights are for the sub fitnesses employed in the GA Engine.

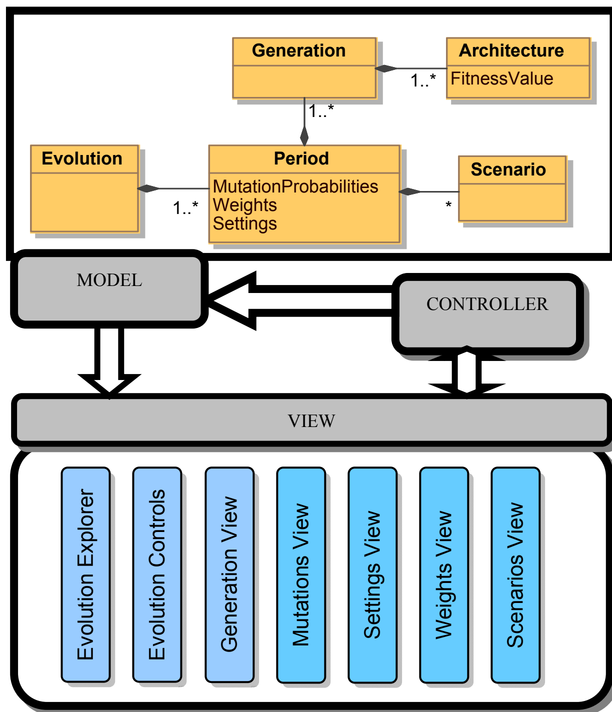


Figure 3. Darwin MVC architecture and simplified conceptual model

The view in our MVC architecture is composed of multiple Eclipse's views. Their purpose will be elucidated in the subsequent section in detail. The controller realizes the entire control logic and keeps the model consistent. Moreover, it also communicates with the Zest, UML2Tools and JFreeChart plugins to perform its operations.

VI. Using the Tool

In this section, first the user interface of the tool is described and then the usage of the tool is illustrated with an example. Here we are using a fairly simple example to facilitate tool presentation; more realistic applications of the approach can be found in our earlier work [23].

A. *Darwin user interface*

Darwin user interface consists of several views, as shown in Fig. 4. In the *evolution explorer*, a user can manage the evolutions in various ways (e.g. creating, opening, saving, removing evolutions etc.). *Evolution controls* are used for starting, pausing, and resuming an evolution (see the upper part of Fig. 4).

The *generation view* shows the individuals present in a generation as shown in Fig. 4. The generation to be viewed can be specified in this view or can be selected directly from the fitness graph. Moreover, the architecture and family tree of an individual can be viewed using this view.

The *mutations view* shows the mutations and their probabilities for a period selected in evolution explorer view. Additionally, it enables the modification of mutation probabilities. Buttons are provided for incrementing and decrementing mutation probability, and for applying default probabilities. The *settings view* is used to alter the population size and generation size of a period. It also provides options to change the fitness calculation method of a period.

In the *weights view*, the weights of the different quality attribute in terms of the fitness function can be determined. In our current realization the fitness function covers modifiability, efficiency, complexity and possible scenarios. The scenarios of a period can be managed in the *scenarios view*. In this view, the user can change the parameters of a scenario, introduce a new scenario, delete an existing scenario, and specify a new set of scenarios from a file.

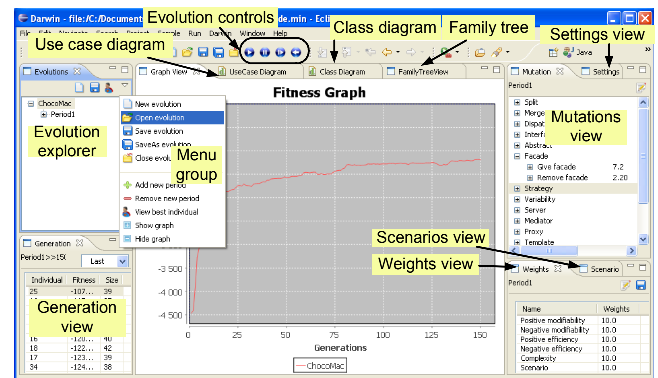


Figure 4. Darwin user interface for an executed evolution

The *fitness graph view* shows the behavior of the fitness function during an evolution: the y-axis gives the fitness

(elite average or best individual) and the x-axis the time (or generation number).

The *family tree view* shows the family tree of the individuals involved in the evolution as a graph. The individuals and their parents are shown in this view, allowing the exploration of the development of a particular family line. The parent relationship is shown as an arc from a child to the parent. Moreover, the family tree can be incrementally extended towards the ancestors.

Class diagram editor is used to give the null architecture of a system and to show the generated architectures as UML class diagrams. The most interesting architecture is of course usually the "result" of the evolution, that is, the best architecture of the last generation. The *use case diagram editor* is used to draw use case diagram and the RDG of a system.

B. Example

Automatic chocolate vending machine (ACVM) is used here as an example system. The first step is to identify the relevant use cases specifying the functional requirements. The use case diagram of the system is given in Fig. 5. Using the ACVM, one can buy desired chocolate using coins, and an administrator can collect the money and also can refill the finished stock.

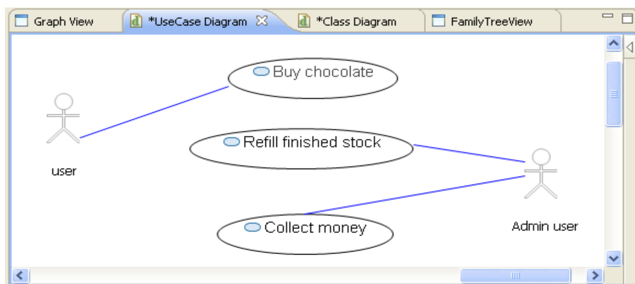


Figure 5. ACVM use case diagram

Next, each use case has to be refined to form an RDG. Here, we have only presented the refinement of the second use case "Refill finished stock" as shown in Fig. 6. We have used packages in use case diagrams to indicate the units (subsystems or components) that own the responsibilities. The entire responsibility set for this system contains 29 functional responsibilities, 8 data responsibilities and 47 dependencies between them.

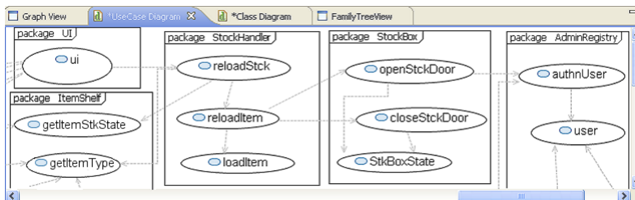


Figure 6. Refining use case "Refill finished stock" to RDG

As the RDG for the system has been produced, the next step is to create the null architecture. The null architecture of the system is created by identifying the logical entities found during use case refinement. As can be seen from Fig. 6, different units are involved in the use case refinement. Each unit will be a component in the null architecture. The resultant null architecture of the system consists of 11 classes (components), and is presented in Fig. 7.

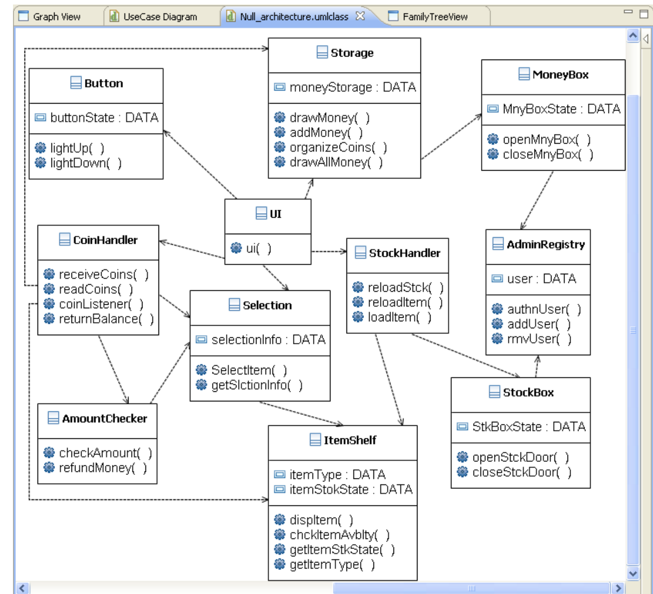


Figure 7. Null architecture for ACVM

To generate candidate architectures for the system, an evolution is first created. Then, the null architecture and RDG of the system are given as input for the evolution. Next, the number of the periods for the evolution and the parameters for each period have to be specified. Here we used one period, with a population of 60 individuals and 90 generations. After some experimentation, suitable mutation probabilities and fitness weights are given to the period. The calculated fitness value for a generation will be the average of fitnesses of 10 best individuals (i.e. elite) of the generation.

Finally, evolution controls are used to apply genetic algorithm on the evolution. As each generation in the evolution is processed, its fitness is drawn on the fitness graph. After observing that the fitness growth is not satisfactory, a new period is included with population size, generation size and weights similar to the existing period, but with different mutation parameters. The resultant fitness graph after 180 generations is presented in Fig. 8. As can be seen, the path of the fitness curve changed after 10th generation. To examine what caused it to change the history of individuals in that generation is explored using the family tree.

A fragment of the family tree of an individual is presented in Fig. 9. The "R" in the family tree corresponds to the rank of the individual in that generation, whereas "OG" implies that the individual is from another generation. As can be seen, the root individual is formed due to crossover operation, while one of its parents is from some other generation and a result of a mutation operation. Moreover, a family tree can be used to examine child and parent architectures.

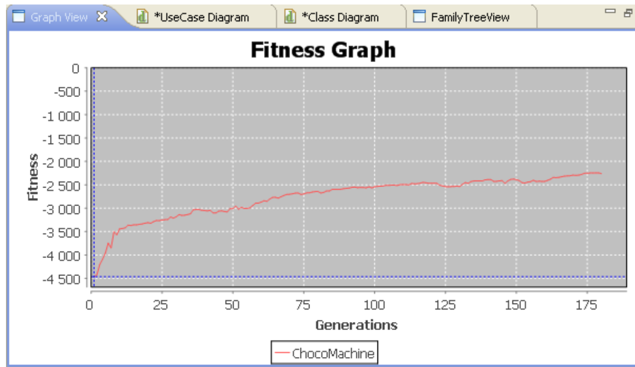


Figure 8. Fitness graph for ACVM

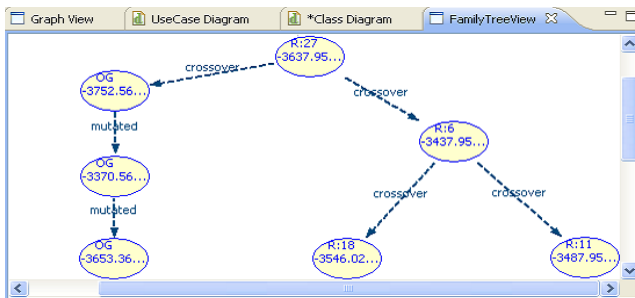


Figure 9. A fragment of family tree of an individual

The best architecture of the last generation can be regarded as the proposed architecture for the ACVM. A part of the proposed architecture for the system is presented in Fig. 10. As can be seen in the figure, certain patterns (adapter, strategy, and template method) have been introduced. The classes related to the introduced patterns are colored in the figure.

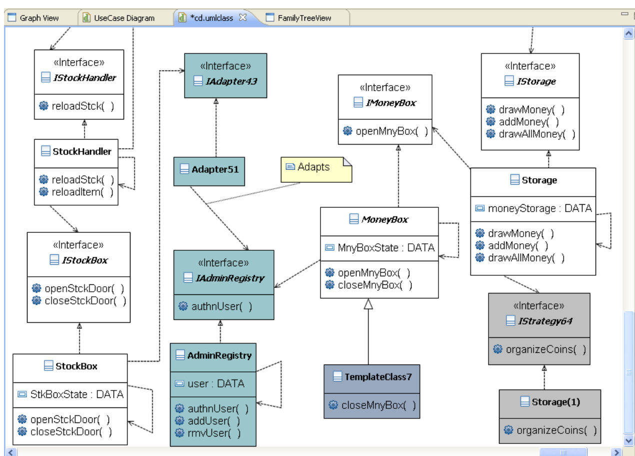


Figure 10. A fragment of proposed architecture for ACVM

For some of the patterns, new classes must be generated, and the real names of such classes cannot be inferred. For example, in the case of Template Method, the name of the required subclass is just “TemplateClass”, with a unifying suffix (e.g. TemplateClass7 in Fig. 10). Another example is Strategy: a new class (and interface) is generated for the strategy method. In this case the new class is named according to the original host class of the method, with a unifying suffix (e.g. Storage(1) in Fig. 10). For Adapter, the adapter class is named simply “Adapter”, with a unifying suffix.

VII. EFFICIENCY

A potential drawback of using the GA approach is performance: the evolution of large populations of complex individuals might be time-consuming. Since we aim at an interactive tool which aids the architect in exploring different kinds of solutions with different parameters, the execution of the evolution should not be too slow.

In our tool, the user can choose between two modes: fast mode with less history recording, and slow mode with full history recording. With fast mode, the execution time of a typical evolution is reasonable, taking into account that the architect can follow the development of the fitness curve in real time and stop the evolution when the fitness is no more improving. The time depends mainly on the size of the population (and naturally on the number of generations). A graph depicting the execution times when the tool was tested using an increasing set of population sizes is shown in Fig. 11. The target system was the same ACVM. The total amounts of generations were fixed at 100 in the tests. The highest value of execution time we observed was 96 seconds when population size had reached 300.

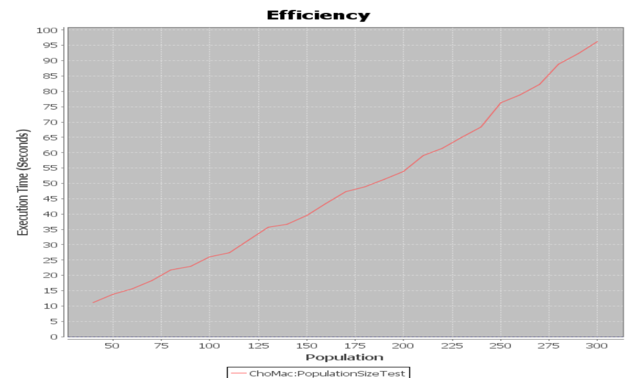


Figure 11. Efficiency with different population sizes

VIII. CONCLUSIONS

Based on our earlier work on applying genetic algorithms for software architecture synthesis, we have devised a process model for developing a software architecture using this approach, and explored the required tool support to assist the process. We have developed a prototype tool called Darwin to test and demonstrate the tool ideas. Given that the ultimate goal is to automate substantial part of software architecture design in practice, the work in this paper represents only first steps.

A necessary requirement for such a tool is tight integration with a conventional design environment with architectural modeling capabilities. Ideally, the tool should be also integrated with a large, constantly growing knowledge base of architectural solutions in a form that can be understood by the GA engine. In this way, the tool could

make use of community knowledge that is virtually impossible to master by a single human. We argue that in such a setup the tool not only automates the design process, but it can in many cases actually outperform a human architect who tends to be confined with a fairly limited set of solutions originating from his or her past experience.

A major topic of our future work is the enabling of the use of existing architectures as input. We are also aiming to implement a support for preserving the given solutions (e.g., architectural styles) throughout an evolution.

ACKNOWLEDGMENT

This work is a part of the Darwin project, funded by the Academy of Finland.

REFERENCES

[1] M. Amoui, S. Mirarab, S. Ansari, and C. Lucas, “A genetic algorithm approach to design evolution using design pattern transformation,” International Journal of Information Technology and Intelligent Computing, vol. 1, 2006, pp. 235-245.

[2] P. Avgeriou, P. Kruchten, P. Lago, P. Grisham, and P. Dewayne, “Architectural Knowledge and Rationale - Issues, Trends, Challenges,” ACM Sigsoft Software Engineering Notes, vol. 32, issue 4, July 2007, pp. 41-46.

[3] F. Bachmann, L. Bass, and M. Klein, “Preliminary design of ArchE: a software architecture design assistant,” Technical Report, CMU/SEI-2003-TR-021, 2003.

[4] W.J. Brown, R.C. Malveau, H.W. McCormickIII, and T.J. Mowbray, Antipatterns - Refactoring Software, Architectures, and Projects in Crisis, Jhon Wiley and Sons, Inc., 1998.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, vol. 1, John Wiley & Sons, August 1996.

[6] S.R. Chidamber and C.F. Kemerer, “A metrics suite for object oriented design,” IEEE Transactions on Software Engineering, vol. 20, issue 6, June 1994, pp. 476-493.

[7] J. Clarke et al., “Reformulating software engineering as a search problem,” IEE Proceedings - Software, vol 150, issue 3, 2003, pp. 161-175.

[8] P. Clements, R. Kazman, and M. Klein, Evaluating Software Architectures, Addison-Wesley, 2002.

[9] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo, “A language independent software renovation framework,” The Journal of Systems and Software, vol. 77, issue 3, 2005, pp. 225-240.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[11] F.W. Glover and G.A. Kochenberger, Handbook of Metaheuristics, International Series in Operations Research & Management Science, 57th ed., Springer, 2003.

[12] A. Jansen, J. Bosch, and P. Avgeriou, “Documenting after the fact: Recovering architectural design decisions,” The Journal of Systems and Software 81, 2008, pp. 536-557.

[13] R. Lutz, “Evolving good hierarchical decompositions of complex systems,” Journal of Systems Architecture, vol. 47, 2001, pp. 613-634.

[14] S. Mancoridis, B.S. Mitchell, C. Rorres, Y.F. Chen, and E.R. Gansner, “Using automatic clustering to produce high-level system organizations of source code,” Proc. International Workshop on Program Comprehension (IWPC 98), USA, 1998, pp. 45-53.

[15] J. McGregor, F. Bachmann, L. Bass, and P. Bianco, “Using ArchE in the classroom: one experience,” Technical Note, CMU/SEI-2007-TN-001, 2007.

[16] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolutionary Programs, Springer-Verlag, 1992.

[17] B.S. Mitchell, S. Mancoridis, and M. Traverso, “Search based reverse engineering,” Proc. 14th International Conference on Software Engineering and Knowledge Engineering (SEKE’02), 2002, pp. 431-438.

[18] M. O’Keeffe and M. Ó Cinnéide, “Towards automated design improvements through combinatorial optimization,” Proc. 4th Internation Workshop on Directions in Software Engineering Environments (WoDiSEE2004), W2S Workshop - 26th International Conference on Software Engineering, 2004, pp. 75 – 82.

[19] M. O’Keeffe and M. Ó Cinnéide, “Search-based software maintenance,” Proc. Conference on Software Maintenance and Reengineering (CSMR'06), 2006, pp. 249-260.

[20] M. O’Keeffe and M. Ó Cinnéide, “Search-based refactoring for software maintenance,” Journal of Systems and Software, vol. 81, issue 4, 2008, pp. 502-516.

[21] O. Räihä, K. Koskimies, and E. Mäkinen, “Genetic Synthesis of Software Architecture,” Proc. 7th International Conference on Simulated Evolution and Learning (SEAL’08), Springer LNCS 5361, 2008, pp. 565-574.

[22] O. Räihä, K. Koskimies, E. Mäkinen, and T. Systä, “Pattern-Based Genetic Model Refinements in MDA,” Nordic Journal of Computing, vol. 14, issue 4, 2008, pp. 322-339.

[23] O. Räihä, K. Koskimies, and E. Mäkinen, “Scenario-based genetic synthesis of software architecture,” Proc. Fourth International Conference on Software Engineering Advances (ICSEA’09), 2009, pp. 437-445.

[24] M. Shaw, and D. Garlan, Software Architecture - Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[25] ArchE WWW site. At URL http://www.sei.cmu.edu/architecture/tools/arche/index.cfm. Checked 15.3.2010.

[26] Eclipse WWW site. At URL http://www.eclipse.org. Checked 11.3.2010

[27] Zest WWW site. At URL http://www.eclipse.org/gef/zest. Checked 11.3.2010

[28] OMG’s UML 2.2 WWW site. At URL http://www.omg.org/technology/documents/formal/uml.htm. Checked 12.3.2010

[29] Eclipse’s Model Development Tools WWW site. At URL http://www.eclipse.org/modeling/mdt . Checked 11.3.2010

[30] JFreeChart’s WWW site. At URL http://www.jfree.org/jfreechart. Checked 10.3.2010.

Paper VI

Outi Räihä, A Survey on Search-Based Software Design. *Computer Science Review*, **4** (4), 2010, 203-249.

Copyright (2010) Elsevier. Reprinted, with permission.

available at www.sciencedirect.com

journal homepage: www.elsevier.com/locate/cosrev

Survey

A survey on search-based software design

*Outi Räihä**

Department of Software Systems, Tampere University of Technology, Korkeakoulunkatu 1, P.O. Box 553, 33101 Tampere, Finland

ARTICLE INFO

Article history:
Received 18 February 2010
Received in revised form
18 June 2010
Accepted 22 June 2010

Keywords:
Search-based software engineering
Software design
Search algorithms
Software quality

ABSTRACT

This survey investigates search-based approaches to software design. The basics of the most popular meta-heuristic algorithms are presented as background to the search-based viewpoint. Software design is considered from a wide viewpoint, including topics that can also be categorized as software maintenance or re-engineering. Search-based approaches have been used in research from the high architecture design level to software clustering and finally software refactoring. Enhancing and predicting software quality with search-based methods is also taken into account as a part of the design process. The background for the underlying software engineering problems is discussed, after which search-based approaches are presented. Summarizing remarks and tables collecting the fundamental issues of approaches for each type of problem are given. The choices regarding critical decisions, such as representation and fitness function, when used in meta-heuristic search algorithms, are emphasized and discussed in detail. Ideas for future research directions are also given.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

Traditional software engineering attempts to find solutions to problems in a variety of areas, such as testing, software design, requirements engineering, etc. A human software engineer must apply his acquired knowledge and resources to solve such complex problems that have to simultaneously meet needs but also be able to handle constraints. Often there are conflicts regarding the wishes of different stakeholders, i.e., compromises must be made with decisions regarding both functional and non-functional aspects. However, as in any other engineering discipline, software engineers still attempt to find the optimal solution to any given problem, regardless of its complexity. As systems get more complex, the task of finding even a near optimal solution will become far too laborious for a human. Automating (or semi-automating) the process of finding, say, the optimal software architecture or resource allocation in a software project, can thus be seen as the ultimate dream in software engineering. Results from applications of search techniques in other engineering disciplines further support this idea, as they have been extremely encouraging.

Search-based software engineering (SBSE) applies meta-heuristic search techniques, such as genetic algorithms and simulated annealing, to software engineering problems. It stems from the realization that many tasks in software engineering can be formulated as combinatorial search problems. The goal is to find, from the wide range of possibilities, a solution that is sufficiently good according to an appropriate quality function. Ideally this would be the optimal solution, but in reality optimality may be difficult (if not impossible) to achieve or even define due to various reasons, such as the size of the search space or the complexity of the quality function. Allowing a search algorithm to find a solution from such a wide space enables partial or full automation of previously laborious tasks, solves problems

* Tel.: +358 50 5342813; fax: +358 3 31152913.
E-mail address: outi.raiha@tut.fi.

1574-0137/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.cosrev.2010.06.001

that are hard to manage by other methods, and often leads to solutions that a human software engineer might not have been able to think of.

Interest in SBSE has been growing rapidly over recent years, both in academia and industry. The combination of increased computing power, and new, more efficient, search algorithms has made SBSE a practical solution method for many problems throughout the software engineering life cycle [1]. Harman [2] has provided a brief overview to the current state of SBSE, and problems in the field of software engineering have been formulated as search problems by Clarke et al. [3] and Harman and Jones [4].

Search-based approaches have been most extensively applied in the field of software testing, and a covering survey of this branch (focusing on test data generation) has been made by McMinn [5]. A review on SBSE, concentrating on testing, is also provided by Mantere and Alander [6]. Another test related survey has been made by Afzal et al. [7,8], who concentrate on testing non-functional properties. As there has been much research and many previous surveys regarding the area of testing, it will be omitted from this survey, even if the studies related to testing could be considered as altering (and thus perhaps improving) a software design. This happens, e.g., with testability transformations. Harman et al. [9] define three critical differences to traditional transformations, one of them concerning the functionality of the program, and state that "testability transformations need not preserve functional equivalence". This contradicts the idea of building a design based on a fixed set of requirements.

This survey will cover the branch of software design. Software design can be defined as "the process which translates the requirements into a detailed design of a software system" [10]. Here software design is considered as described by Wirfs-Brock and Johnson [11]. Although they consider only object-oriented design, the skeleton of a process from requirements to actual design can be applied to any form of software design. A design process starts from requirements, and first enters an exploratory phase, where the fundamental structure is decided. This leads to a preliminary design, which then enters an analysis stage. After the suggested design is analyzed and modified according to the result, the final design is achieved. Following this interpretation, software refactoring and clustering have also been taken into account, as they are considered as actions of modifying (based on a certain analysis) a preliminary model, which in many cases is a working implementation.

The area of search-based software design has developed greatly in very recent years, and is gaining an increasing interest in the SBSE community. Although several surveys have been made of the SBSE field as a whole, they deal with the design area quite briefly. Also, the literature published from the software design perspective either does not cover search-based methods [10,12] or only briefly mentions the option of having an algorithm to automate class hierarchy design [11]. Thus, there is a need to cover this crossing of two disciplines: search-based techniques and software design. A new contribution is made, especially in summarizing research in architecture level design that uses search-based techniques, as this has been overlooked in previous studies of search-based software engineering.

A special note should be made of the value added to a recent comprehensive review on SBSE made by Harman et al. [13], who give a thorough but very compact view of the field as a whole. As they cover a very great number of references, and the main contributions of the survey (as stated by Harman et al. [13]) are coverage and completeness, classification and trend analysis, it is natural that the presentation of the papers lacks in some depth. The actual studies are presented as they are, without criticism or discussion of the particularities of a certain technique, although basic information of each technique is collected in categorical tables. In particular, the area of software design on the architecture level is very briefly dealt with.

To this end, the present survey adds to the contribution of the survey of Harman et al. [13] by giving a thorough view of research in the area of search-based software design. The area of software architecture design is given special attention, and some additional recent references are also included. The presented papers are discussed in detail and critically analyzed. Summarizing remarks on the similarities and differences between techniques are also provided.

Additionally, as Harman and Wegener [14] point out, choosing the representation and fitness function is crucial in all search-based approaches to software engineering. When using genetic algorithms [15], which are especially popular in search-based design, the choices regarding genetic operators are just as important and very difficult to make. Thus, this survey emphasizes the choices made regarding the particular characteristics of search algorithms. The small but critical decisions, such as what fitness function, encoding and operations to use, are discussed and categorized in detail. This helps in easily finding the distinct differences between similar techniques, and identifying best practices. Also, any new study in the field of search-based software engineering would benefit from learning what kind of solutions have proven to be particularly successful in the past.

The timeline for development of SBSE as a field is presented in Fig. 1. It can clearly be seen that the earliest applications were in testing, as can be deduced from the number of existing surveys. However, more importantly, the timeline also shows the steady increase of ideas in the area of search based design in the past 10 years. Thus, a survey covering this area is certainly due. All in all, the timeline shows that SBSE has been a very active discipline in the past 20 years, as only novel ideas are presented here. Countless approaches and studies regarding these ideas have been made but are not portrayed here. The explanations and references for the data points in Fig. 1 are given in Table 1.

This survey proceeds as follows. Section 2 describes search algorithms; and the underlying concepts for genetic algorithms, simulated annealing and hill climbing are discussed in detail. Different ways of performing the exploratory phase of design are then presented as methods for software architecture design (object-oriented and service-oriented) in Section 3. Sections 4–6 deal with clustering, refactoring and software quality, respectively, all of which can be seen as components of the analysis phase, starting from higher level re-design (clustering), going to low-level re-design (re-factoring) and finally pure analysis. The background for each underlying problem is first presented, followed by recent approaches applying search-based techniques to the problem. Summarizing remarks and a summary table of the studies is presented after each subsection. Finally, some ideas for future work are given in Section 7, and conclusions are presented in Section 8.

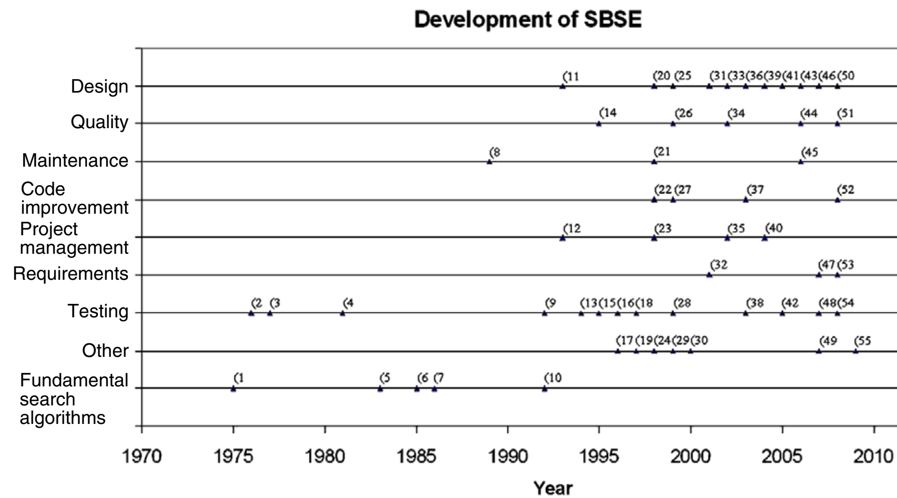


Fig. 1 – Timeline of SBSE development.

Table 1 – References for timeline data points.

No.	Description
(1)	Genetic algorithm (GA) [15]
(2)	Test data automation [110]
(3)	Test case automation [111]
(4)	Retesting [112]
(5)	Simulated annealing (SA) [113]
(6)	Genetic programming (GP) [114]
(7)	Tabu search [115]
(8)	Revalidation [116]
(9)	GAs in testing [117]
(10)	Ant colony optimization (ACO) [99]
(11)	GA, constraints [118]
(12)	GA, project management [119]
(13)	GA, test data [120]
(14)	GA, reliability model [121]
(15)	Chaining approach, test data [122]
(16)	GA, structural testing [123]
(17)	GA, protocol validation [124]
(18)	GA, response time [125]
(19)	GA, GP, software agents [126]
(20)	Clustering [64]; parallelization [93]
(21)	GP, software versioning [127]
(22)	SA, flaw finding [128]
(23)	Project estimation [129]
(24)	Compiler [130]; Task scheduling [131]
(25)	GP, re-engineering at code level [132]
(26)	GP, quality determination [133]
(27)	GA, reduced code space [134]
(28)	SA, regression testing [135]
(29)	GA, Protocols for distributed applications [136]
(30)	Secure protocols [137]
(31)	GA, hierarchical decompositions [42]
(32)	Next release problem [138]
(33)	GA, reverse engineering at architecture level [139]
(34)	GA, combining quality predictive models [105]
(35)	GP, project effort estimation [140]
(36)	Multiple hill climbing, clustering [72]
(37)	GA, code transformations [141]
(38)	SA, test suites [142]
(39)	Architecture relations [143]; service composition [59]
(40)	Project resource allocation [144]
(41)	Amorphous slicing [145]
(42)	ACO, testing [146]
(43)	GA, design patterns [32]
(44)	SA, quality prediction [107]
(45)	GA, software integration [147]
(46)	Use case -based design [29,30]; GA, repackaging [38]; Pareto optimal refactoring [95]
(47)	Multiobjective next release problem [148]
(48)	ACO, model checking [149]; GP, model checking [150]; Pareto optimality, test cases [151]
(49)	GA, code author identification [152]
(50)	Class responsibility assignment [28]; Software behavior modeling [40]; Architecture design [33]; model transformations by GA [34] and particle swarm optimization [36]
(51)	Software verification [153]
(52)	Co-evolution, bug fixing [154]
(53)	Requirements optimization [155]
(54)	Tabu search, testing [156]
(55)	GA, decision making in autonomic computing systems [157]

2. Search algorithms

Meta-heuristics are commonly used for combinatorial optimization, where the search space can become especially large. Many practically important problems are NP-hard, and thus, exact algorithms are not possible. Heuristic search algorithms handle an optimization problem as a task of finding a "good enough" solution among all possible solutions to a given problem, while meta-heuristic algorithms are able to solve even the general class of problems behind the certain problem. A search would optimally end in a global optimum in a search space, but at the very least it will give some local optimum, i.e., a solution that is "better" than a significant amount of alternative solutions nearby. A solution given by a heuristic search algorithm can be taken as a starting point for further searches or be taken as the "best" possible solution, if

its quality is considered high enough. For example, simulated annealing can be used to produce seed solutions for a genetic algorithm that constructs the initial population based on the provided seeds.

In order to use search algorithms in software engineering, the first step is that the particular software engineering problem should be defined as a search problem. If this cannot be done, search algorithms are most likely not the best way to solve the problem, and defining the different parameters and operations needed for the search algorithm can be difficult. After this has been done, a suitable algorithm can be selected and the issues regarding that algorithm must be dealt with.

There are three common issues that need to be dealt with by any search algorithm: 1. encoding the solution, 2. defining transformations, and 3. measuring the "goodness" of a solution. All algorithms need the solution to be encoded according to the algorithm's specific needs. For example, in order for the genetic algorithm (GA) to operate, the encoding should be done in such a way that it can be seen as a chromosome consisting of a set of genes. However, for the hill climbing (HC), any encoding where a neighborhood can be defined is sufficient. The importance and difficulty of encoding a solution increase as the complexity of the problem at hand increases. In this case complexity refers to how easily a solution can be defined, rather to the computational complexity of the problem itself. For example, a job-shop problem may be computationally complex, but the solution candidates are simple to encode as an integer array. However, a solution containing, e.g., all the information regarding a software architecture, is demanding to encode so that: 1. all information stays intact, 2. operations can efficiently be applied to the selected encoding of the solution, 3. the fitness evaluations can be performed efficiently, and 4. there is minimal need for "outside" data, i.e., data structures containing information about the solution that are not included in the actual encoding.

Defining a neighborhood is crucial to all algorithms; HC, simulated annealing (SA) and tabu search operate purely on the basis of moving from one solution to its neighbor. A neighbor is achieved by some operation that transforms the solution. These operations can be seen as equivalent to the mutations needed by the GA.

Finally, the most important and difficult task is defining a fitness function. If defining the fitness function fails, the search algorithm will not be guided towards the desired solutions. All search algorithms require this quality function to evaluate the "goodness" of a solution in order to compare solutions and thus guide the search.

To understand the basic concepts behind the approaches presented here, the most commonly used search algorithms are briefly introduced. The most common approach is to use genetic algorithms. Hill climbing and its variations, e.g., multi-ascent hill climbing (MAHC), is also quite popular due to its simplicity. Finally, several studies use simulated annealing. In addition to these algorithms, tabu search is a widely known meta-heuristic search technique, and genetic programming (GP) [16] is commonly used in problems that can be encoded as trees. For a detailed description on GA, see [17] or [18], for SA, see, e.g., [19], and for HC, see [20], who also cover a wide range of other meta-heuristics. For a description on multi-objective optimization with evolutionary algorithms, see [21] or [22]. A survey on model-based search, covering several meta-heuristic algorithms is also made by Zlochin et al. [23].

2.1. Genetic algorithms

Genetic algorithms were invented by John Holland in the 1960s. Holland's original goal was not to design application specific algorithms, but rather to formally study the ways of evolution and adaptation in nature and develop ways to import them into computer science. Holland [15] presents the genetic algorithm as an abstraction of biological evolution and gives the theoretical framework for adaptation under the genetic algorithm [17].

In order to explain genetic algorithms, some biological terminology needs to be clarified. All living organisms consist of cells, and every cell contains a set of chromosomes, which are strings of DNA and give the basic information of the particular organism. A chromosome can be further divided into genes, which in turn are functional blocks of DNA, each gene representing some particular property of the organism. The different possibilities for each property, e.g., different colors of the eye, are called alleles. Each gene is located at a particular locus of the chromosome. When reproducing, crossover occurs: genes are exchanged between the pair of parent chromosomes. The offspring is subject to mutation, where single bits of DNA are changed. The fitness of an organism is the probability that the organism will live to reproduce and carry on to the next generation [17]. The set of chromosomes at hand at a given time is called a population.

Genetic algorithms are a way of using the ideas of evolution in computer science. When thinking of the evolution and development of species in nature, in order for the species to survive, it needs to develop to meet the demands of its surroundings. Such evolution is achieved with mutations and crossovers between different chromosomes, i.e., individuals, while the fittest survive and are able to participate in creating the next generation.

In computer science, genetic algorithms are used to find a good solution from a very large search space, the goal obviously being that the found solution is as good as possible. To operate with a genetic algorithm, one needs an encoding of the solution, i.e., a representation of the solution in a form that can be interpreted as a chromosome, an initial population, mutation and crossover operators, a fitness function and a selection operator for choosing the survivors for the next generation. Algorithm 1 gives the pseudo code for a genetic algorithm.

Algorithm 1 geneticAlgorithm

```
Input: formalization of solution, initialSolution
  chromosomes ← createPopulation(initialSolution)
  while NOT terminationCondition do
  foreach chromosome in chromosomes
    p ← randomProbability
    if p > mutationProbability then
    mutate(chromosome)
    end if
  end for
  foreach chromosomePair in chromosomes
    cp← randomProbability
    if cp > crossoverProbability then
      crossover(chromosomePair)
      addOffspringToPopulation()
      end if
    end for
    foreach chromosome in chromosomes
      calculatefitness(chromosome)
    end for
    selectNextPopulation()
  end while
```

As discussed, correctly defining the different operations (mutations, crossover and fitness function) is vital in order to achieve satisfactory results. However, as seen in Algorithm 1, there are also many parameters regarding the GA that need to be defined and greatly affect the outcome. These parameters are the population size, number of generations (often used as the terminating condition) and the mutation and crossover probabilities. Having a large enough population ensures variability within a generation, and enables a wide selection of different solutions at every stage of evolution. However, at a certain point the results start to converge, and a larger population always means more fitness evaluations and thus requires more computation time. Similarly, the more generations the algorithm is allowed to evolve for, the higher the chances are that it will be able to reach the global optimum. However, again, letting an algorithm run for, say, 10 000, generations will most probably not be beneficial, as if the operations and parameters have been chosen correctly, a reasonably good optimum should have been found much earlier. Mutation and crossover probabilities both affect how fast the population evolves. If the probabilities are too high, there is the risk that the implementation of genetic operations becomes random instead of guided. Vice versa, if the probabilities are too low there is the risk that the population will evolve too slowly, and no real diversity will exist. A theory to be noted with genetic operators is the building block hypothesis, which states that a genetic algorithm combines a set of sub-solutions, or building blocks, to obtain the final solution. The sub-solutions that are kept over the generations generally have an above-average fitness [Salomon, 1998]. The crossover operator is especially sensitive to this hypothesis, as an optimal crossover would thus combine two rather large building blocks in order to produce an offspring with a one-point crossover.

2.2. *Simulated annealing*

Simulated annealing is originally a concept in physics. It is used when the cooling of metal needs to be stopped at given points, at which the metal needs to be warmed a bit, before resuming the cooling process. The same idea can be used to construct a search algorithm. At a certain point of the search, when the fitness of the solution in question is approaching a set value, the algorithm will briefly stop the optimizing process and revert to choosing a solution that is not the best in the current solution's neighborhood. This way getting stuck to a local optimum can effectively be avoided. Since the fitness function in simulated annealing algorithms should always be minimized, it is usually referred to as a cost function [19].

Simulated annealing usually begins with a point x in the search space that has been achieved through some heuristic method. If no heuristic can be used, the starting point will be chosen randomly. The cost value c, given by cost function E, of point x is then calculated. Next a neighboring value x_1 is searched and its cost value c_1 calculated. If $c_1 < c$, then the search moves onto x_1. However, even though $c \leq c_1$, there is still a chance, given by probability p, that the search is allowed to continue to a solution with a bigger cost [3]. The probability p is a function of the change in cost function ΔE, and a parameter T:

$$p = e^{-\Delta E/T}.$$

This definition for the probability of acceptance is based on the law of thermodynamics that controls the simulated annealing process in physics. The original function is

$$p = e^{-\Delta E/kT},$$

where T is the temperature in the point of calculation and k is Boltzmann's constant [19].

The parameter T that substitutes the value of temperature and the physical constant is controlled by a cooling function C, and it is very high in the beginning of simulated annealing and is slowly reduced while the search progresses [4]. The actual cooling function is application specific.

If the probability p given by this function is above a set limit, then the solution is accepted even though the cost increases. The search continues by choosing neighbors and applying the probability function (which is always 1 if the cost decreases) until a cost value is achieved that is satisfactorily low. Algorithm 2 gives the pseudo code for a simulated annealing algorithm.

Algorithm 2 simulatedAnnealing

Input: formalization of solution, *initialSolution*, cooling ratio α, initial temperature T_0, frozen temperature T_f, and temperature constant r
Output: optimized solution *finalSolution*
initialQuality $\leftarrow$ evaluate(*initialSolution*) $\leftarrow$ *initialSolution*
$Q_1 \leftarrow$ *initialQuality*
$T \leftarrow T_0$
while $T_0 > T_f$ **do**
$r_i \leftarrow 0$
while $r_i > r$ **do**
$S_i \leftarrow$ findNeighbor(S_1)
$Q_i \leftarrow$ evaluate(Q_1)
if $Q_i > Q_1$ **then**
$S_1 \leftarrow S_i$
$Q_1 \leftarrow Q_i$
else
δ
$\leftarrow Q_1 - Q_{i'}$
$p \leftarrow$ randomProbability
if $p < e^{-\delta/T}$ **then**
$S_1 \leftarrow S_i$
$Q_1 \leftarrow Q_i$
end if
end if'
$r_i \leftarrow r_i + 1$
end while
$T \leftarrow T_{*}\alpha$
end while
return S_1

The key parameters to be adjusted for SA are the initial temperature, the cooling ratio and the temperature constant. The combined effect of these determines how fast the cooling happens. If the cooling is too fast, the algorithm may not have sufficient time to achieve an optimum. However, if the cooling is too slow, the initial temperature may need a significantly high value so that the solution will be able to evolve enough (i.e., noticeably transform from the initial solution) before reaching the frozen temperature.

2.3. *Hill climbing*

Hill climbing begins with a random solution, and then begins to search through its neighbors for a better solution. There are several versions of how this is done; in some versions the algorithm moves on after finding the first neighbor that is better than the current, some do a fixed number of neighbor evaluations and continue to the best of this group, and some versions go through the entire neighborhood of a solution and select the best neighbor from which the procedure is continued. Algorithm 3 adopts the last option, i.e., the entire neighborhood is evaluated before moving on. Hill climbing does not include any mechanisms to avoid getting stuck with a local optimum.

There are three critical choices regarding HC: 1. defining a neighborhood for each solution, 2. defining an evaluation function for a solution, and 3. defining to what extent each neighborhood is searched. If the problem at hand is very complex and each solution has an exponential number of neighbors, traversing through each neighborhood maybe extremely time consuming. However, if the subgroup of neighbors to be examined is chosen wisely, the actual outcome of the algorithm may still be good enough, while much time is saved when not every solution needs to be evaluated.

Algorithm 3 hillClimbing

Input: formalization of solution, *initialSolution*
currentSolution $\leftarrow$ *initialSolution*
currentFitness $\leftarrow$ evaluate(*currentSolution*)
while betterNeighborsExist **do**
neighborhood $\leftarrow$ findNeighbors(*currentSolution*)
foreach *neighbor* **in** *neighborhood*
neighborFitness $\leftarrow$ evaluate(*neighbor*)
if *neighborFitness* > *nextFitness* **then**
nextSolution $\leftarrow$ *neighbor*
nextFitness $\leftarrow$ *neighborFitness*
end if
end for
if *nextFitness* > *currentFitness* **then**
currentSolution $\leftarrow$ *nextSolution*
else
termination
return currentSolution
end if
end while

3. Software architecture design

The core of every software system is its architecture. Designing software architecture is a demanding task requiring much expertise and knowledge of different design alternatives, as well as the ability to grasp high-level requirements and piece them together to make detailed architectural decisions. In short, designing software architecture takes verbally formed functional and quality requirements and turns them into some kind of formal model that is used as a base for code. Automating the design of software is obviously a complex task, as the automation tool would need to understand intricate semantics, have access to a wide variety of design alternatives, and be able to balance multi-objective quality factors. From the re-design perspective, program comprehension is one of the most expensive activities in software maintenance. The following sections describe meta-heuristic approaches to software architecture design for object-oriented and service-oriented architectures.

3.1. *Object-oriented architecture design*

3.1.1. *Background*

At its simplest, object-oriented design deals with extracting concepts from, e.g., use cases, and deriving methods and attributes, which are distributed into classes. A further step

is to consider interfaces and inheritance. A final design can be achieved through the implementation of architecture styles [24] and design patterns [25]. When attempting to automate the design of object-oriented architecture from the concept level, the system requirements must be formalized. After this, the major problem lies within quality evaluation, as many design decisions improve some quality attributes [26] but weaken others. Thus, a sufficient set of quality estimators should be used, and a balance should be found between them. Re-designing software architectures automatically is slightly easier than building architecture from the very beginning, as the initial model already exists and it merely needs to be ameliorated. However, implementing design patterns is never straightforward, and measuring their impact on the quality of the system is difficult. For more background on software architectures, see, e.g., [27].

Approaches to search-based software design are presented in Section 3.1.2 starting from low-level approaches, i.e., what is needed when first beginning the architecture design, to high-level approaches, ending with analyzing software architecture. Object-oriented architecture design begins with use cases and assigning responsibilities, i.e., methods and attributes to classes [29,30,28]. After the basic structure, the architecture can be further designed, either by applying design patterns on an existing system [32] or by building the design patterns into the system from the very beginning [33–35]. If an idea for an optimal solution is available, model transformations can be sought to achieve that solution [36]. There might also be many choices regarding the components of the architecture, depending on the needs of the system. An architecture can be made of alternative components [37] or a subsystem can be sought after [38]. Studies have also been made on identifying concept boundaries and thus automating software comprehension [39], and composing behavioral models for autonomic systems [40,41], which give a dynamic view of software architecture. One of the most abstract studies attempts to build hierarchical decompositions for a software system [42], which already comes quite close to software clustering. Summarizing remarks of the approaches are given in Section 3.1.3, and the fundamentals of each study are collected in Table 2.

3.1.2. *Approaches*

Bowman et al. [28] study the use of a multi-objective genetic algorithm (MOGA) in solving the class responsibility assignment problem. The objective is to optimize the class structure of a system through the placement of methods and attributes. The strength Pareto approach (SPEA2) is used, which differs from a traditional GA in containing an archive of individuals from past populations. This approach combines several aspects that aid in finding the truly optimal individuals and thus leaves less room for GA "to err" in terms of undesired mutations or overly relying on metrics.

The chromosome is represented as an integer vector. Each gene represents a method or an attribute in the system and the integer value in a gene represents the class to which the method or attribute in that locus belongs. Dependency information between methods and attributes is stored in a separate matrix. Mutations are performed by simply changing the class value randomly; the creation of new classes is also allowed. Crossover is the traditional one-point one. There are also constraints: no empty classes are allowed (although the selected encoding method also makes them impossible), conceptually related methods are only moved in groups, and classes must have dependencies on at least one other class.

The fitness function is formed of five different values measuring cohesion and coupling: 1. method–attribute coupling, 2. method–method coupling, 3. method–generalization coupling, 4. cohesive interaction and 5. ratio of cohesive interaction. A complementary measure for common usage is also used. Selection is made with a binary-tournament selection, where the fitter individual is selected 90% of the time.

In the case study an example system is used, and a high-quality UML class diagram of this system is taken as a basis. Three types of modifications are made and finally the modifications are combined in a final test. The efficiency of the MOGA is now evaluated in relation to how well it fixed the changes made to the optimal system. Results show that in most cases the MOGA managed to fix the made modifications and in some cases the resulting system also had a higher fitness value than the original "optimal" system.

Bowman et al. [28] also compare MOGA to other search algorithms, such as random search, hill climbing and a simple genetic algorithm. Random search and hill climbing only managed to fix a few of the modifications, and the simple GA did not manage to fix any of the modifications. Thus, it would seem that a more complex algorithm is needed for the class responsibility assignment problem.

The need for highly developed algorithms is further highlighted when noting that a ready system is being ameliorated instead of completely automating the class responsibility assignment. As a ready system can be assumed to have some initial quality, and conceptually similar methods and attributes are already largely grouped, it does help the algorithm when re-assigning the moved methods and attributes. This is due to the fact that by attempting to re-locate the moved method or attribute to the "wrong" class, the fitness value will be significantly lower than when assigning the method or attribute to the "right" class.

Simons and Parmee [29–31] take use cases as the starting point for system specification. Data is assigned to attributes and actions to methods, and a set of uses is defined between the two sets. The notion of class is used to group methods and attributes. Each class must contain at least one attribute and at least one method. Design solutions are encoded directly into an object-oriented programming language. This approach starts with pure requirements and leaves all designing to the algorithm, making the problem of finding an optimal class structure very much more difficult than in cases where a ready system can be used as basis.

A single design solution is a chromosome. In a mutation, a single individual is mutated by locating an attribute and a method from one class to another. For crossover, two individuals are chosen at random from the population and their attributes and methods are swapped based on their class position within the individuals. Cohesiveness of methods (COM) is used to measure fitness; fitness for class C is defined as $f(C) = 1/(|Ac||Mc|) * \sum(\Delta_{ij})$, where Ac (respectively Mc) stands for the number of attributes

Table 2 – Studies in search-based object-oriented software architecture design.

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Bowman et al. [28]	Class structure design is (semi-) automated	Class diagram as methods, attributes and associations	Integer vector and a dependency matrix	Randomly change the class of method or attribute	Standard one-point	Cohesion and coupling	Optimal class structure	Comparison between different algorithms
Simons and Parmee [29–31]	Class structure design is automated	Use cases; data assigned to attributes and actions to methods	A design solution where attributes and methods are assigned to classes	An attribute and a method are moved from one class to another	Attributes and methods of parents are swapped according to class position	Cohesiveness of methods (COM)	Basic class structure for system.	Design solutions encoded directly into a programming language
Amoui et al. [32]	Applying design patterns; high level architecture design	Software system	Chromosome is a collection of supergenes, containing information of pattern transformations	Implementing design patterns	Single-point crossovers for both supergene level and chromosome level, with corrective function	Distance from main sequence	Transformed system, design patterns used as transformations to improve modifiability	New concept of supergene used
Räihä et al. [33]	Automating architecture design	Responsibility dependency graph	Chromosome is a collection of supergenes, containing information of responsibilities and design patterns	Mutations apply architectural design patterns and styles	A standard one-point crossover with corrective function	Efficiency, modifiability and complexity	UML class diagram depicting the software architecture	
Räihä et al. [34]	Automating CIM-to-PIM model transformations	Responsibility dependency graph and domain model (CIM model)	Chromosome is a collection of supergenes, containing information of responsibilities and design patterns	Mutations apply architectural design patterns and styles	A standard one-point crossover with corrective function	Efficiency, modifiability and complexity	UML class diagram depicting the software architecture (PIM model)	
Räihä et al. [35]	Automating architecture design	Responsibility dependency graph and domain model	Chromosome is a collection of supergenes, containing information of responsibilities and design patterns	Mutations apply architectural design patterns and styles	A standard one-point crossover with corrective function	Efficiency, modifiability, complexity and modifiability related scenarios	UML class diagram depicting the software architecture	

Table 2 (continued)

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Kessentini et al. [36]	Using PSO for model transformations	Source model, target model and mapping blocks	Integer vector	N/A	N/A	Number of source model constructs that can be transformed	Optimal transformations	Particle Swarm Optimization (PSO) used as search algorithm
Kim and Park [37]	Dynamic selection of software components	Softgoal interdependency graph, decision variables	String of integers representing decision variables	Goes through each gene and changes the digit according to mutation probability	Two-point crossover	Quality attributes given by user	Optimal architectural instance from the set of all instances	
Bodhuin et al. [38]	Automating class clustering in jar archives	A grouping of classes of a system	An integer array, each gene is a cluster of classes allocated to the jar represented by integer	Changes the allocation of a class cluster to another jar archive	Standard one-point	Download cost of jar archive	Optimal packaging; finding the subsets of classes most likely to be used together (to be placed in same jar archive)	
Gold et al. [39]	Using GA in the area of concepts	Hypothesis list for concepts	One or more segment representations	A hypothesis location is randomly replaced within a segment pair	Segment pairs of overlapping locations are combined, rest copied	Strongest evidence for segments and hypothesis binding	Optimized concept assignment	Hill climbing used as well as GA
Goldsby and Chang [40]; Goldsby et al. [41]	Designing a system from a behavioral point of view	A class diagram, optional state diagram	A set of behavioral instructions	Changes, removes or adds an instruction	Self-replication	Number of executed tasks	UML state diagram giving the behavioral model of system	No actual evolutionary algorithm used, but a platform that is "an instance of evolution"
Lutz [42]	Information theory applied in software design; high-level architecture design	Software system	Hierarchical modular decomposition (HMD)	Three mutations operating the module tree for the HMD	A variant of tree-based crossovers, as used in GP, with corrective function	1/complexity	Optimal hierarchical decomposition of system	

(respectively methods) in class C, and $\Delta_{ij} = 1$, if method j uses attribute i, and 0 otherwise. Selection is performed by tournament and roulette-wheel. The choices regarding encoding, genetic operators and fitness function are quite traditional, although the problem to be solved is far from traditional.

In an alternative approach, categorized by the authors as *evolutionary programming (EP)* and inspired by Fogel et al. [43], offspring are created by mutation and selection is made with tournament selection. Two types of mutations are used, class-level mutation and element-level mutation. At the class level, all attributes and methods of a class in an individual are swapped as a group with another class selected at random. At the element level, elements (methods and attributes) in an individual are swapped at random from one class to another. Initialization of the population is made by allocating a number of classes to each individual design at random, within a range derived from the number of attributes and methods. All attributes and methods from sets of attributes and methods are then allocated to classes within individuals at random. These operations appear quite simplistic, and the actual change to the design remains minimal, since the fitness of an individual depends on how methods and attributes depending on one another are located. When the elements are moved in a group, there does not seem to be very much change in the actual design.

A case study is made with a cinema booking system with 15 actions, 16 data and 39 uses. For GA, the average COM fitness for the final generation for both tournament and roulette-wheel is similar, as is the average number of classes in the final generation. However, convergence to a local optimum is quicker with tournament selection. Results reveal that the average and maximum COM fitness of the GA population with roulette-wheel selection lagged behind tournament in terms of generation number. For EP, the average population COM fitness in the final generation is similar to that achieved by the GA.

The initial average fitness values of the three algorithms are notably similar, although the variance of the values increases from GA tournament to GA roulette-wheel to EP. In terms of COM cohesion values, the generic operators produced conceptual software designs of similar cohesion to human performance. Simons and Parmee [29–31] suggest that a multi-objective search may be better suited for support of the design processes of the human designer. To take into account the need for extra input, they attempted to correct the fitness function by multiplying the COM value by (a) the number of attributes and methods in the class (COM.M+A); (b) the square root of the number of attributes and methods in the class (COM.$\sqrt{(M + A)}$; (c) the number of uses in the class (COM.uses) and (d) the square root of the number of uses in a class (COM. $\sqrt{}$uses). Using such multipliers raises some questions, as there is no intuition for using the square root multipliers. Multiplying by the sum of methods and attributes or uses can intuitively be justified by showing more appreciation to classes that are large but are still comprehensible. However, such an appreciation may lead to preferring larger classes.

The authors have taken this into account by measuring the number of classes in a design solution, and a design solution with a higher number of classes is preferred to a design solution with fewer classes. When cohesion metrics that take class size into account are used, there is a broad similarity between the average population cohesion fitness and the manual design. Values achieved by the COM.M+A and COM.uses and cohesion metrics are higher than the manual design cohesion values, while COM.$\sqrt{(M + A)}$ and COM.$\sqrt{}$uses values are lower. Manually examining the design produced by the evolutionary runs, a difference is observed in the design solutions produced by the four metrics that account for class size, when compared with the metrics that do not. From the results produced for the two case studies, it is evident that while the cohesion metrics investigated have produced interesting cohesive class design solutions, they are by no means a complete reflection of the inherently multi-objective evaluations conducted by a human designer. The evolutionary design variants produced are thus highly dependent on the extent and choice of metrics employed during search and exploration. These results further emphasize the importance of properly defining a fitness function and deciding on the appropriate metrics in all software design related problems.

Amoui et al. [32] use the GA approach to improve the reusability of software by applying architecture design patterns to a UML model. The authors' goal is to find the best sequence of transformations, i.e., pattern implementations. Used patterns come from the collection presented by Gamma et al. [25], most of which improve the design quality and reusability by decreasing the values of diverse coupling metrics while increasing cohesion.

Chromosomes are an encoding of a sequence of transformations and their parameters. Each individual consists of several *supergene*s, each of which represents a single transformation. A supergene is a group of neighboring genes on a chromosome which are closely dependent and are often functionally related. Only certain combinations of the internal genes are valid. Invalid patterns possibly produced through mutations or crossover are found and discarded. The supergene concept introduced here is an insightful approach into handling masses of complex data that needs to be represented as a relatively simple form. Instead of having only one piece of information per gene, this way several pieces of related information can be grouped to such supergenes, which then logically form a chromosome. In the study by Bowman et al. [28] the need for additional data storage (the matrix for data dependencies) demonstrates the complexity of design problems. In this case the supergene approach introduced by Amoui et al. [32] could have been worthwhile trying to include all information regarding the attributes and methods in the chromosome encoding.

Mutation randomly selects a supergene and mutates a random number of genes inside the supergene. After this, validity is checked. In the case of encountering a transformed design which contradicts object-oriented concepts, for example, a cyclic inheritance, a zero fitness value is assigned to the chromosome. This is an interesting way of dealing with anomalies; instead of implementing a corrective operation to force validity, it is trusted that the fitness function will suffice in discarding the unsuitable individuals if they are given a low enough value.

Two different versions of crossover are used. The first is a single-point crossover applied at the supergene level, with a randomly selected crossover point, which swaps the supergenes beyond the crossover point, while the internal genes of supergenes remain unchanged. This combines the promising patterns of two different transformation sequences. The second crossover randomly selects two supergenes from two parent chromosomes, and similarly applies single point crossover to the genes inside the supergenes. This combines the parameters of two successfully applied patterns. The first crossover thus attempts to preserve high-level building blocks, while the second version attempts to create low-level building blocks.

The quality of the transformed design is evaluated, as introduced by Martin [44], by its "distance from the main sequence" (*D*), which combines several object-oriented metrics by calculating the abstract classes' ratio and coupling between classes, and measures the overall reusability of a system.

A case study is made with a UML design extracted from some free, open source applications. The GA is executed in two versions. In one version only the first crossover is applied and in second both crossovers are used. A random search is also used to see if the GA outperforms it. Results demonstrate that the GA finds the optimal solution much more efficiently and accurately. From the software design perspective, the transformed design of the best chromosomes are evolved so that abstract packages become more abstract and concrete packages in turn become more concrete. The results suggest that GA is a suitable approach for automating object-oriented software transformations to increase reusability. As the application of design patterns is by no means an easy task, these initial results suggest that at least the structure and needs of the GA do not restrict the automated design of the software architecture.

Räihä et al. [33] take the design of software architecture a step further than Simons and Parmee [29], by starting the design from a responsibility dependency graph. The graph can also be achieved from use cases, but the architecture is developed further than the class distribution of actions and data. A GA is used for the automation of design.

In this approach, each responsibility is represented by a supergene, and a chromosome is a collection of supergenes. The supergene contains information regarding the responsibility, such as dependencies of other responsibilities, and evaluated parameters such as execution time and variability. Here the notion of supergene [32] is efficiently used in order to store a large number of different types of data pieces within the chromosome. Mutations are implemented as adding or removing an architectural design pattern [25] or an interface, or splitting or joining class(es). Implemented design patterns are Façade and Strategy, as well as the message dispatcher architecture style [24]. Dynamic mutation probabilities are used to encourage the application of basic design choices (the architectural style(s)) at the beginning and more refined choices (such as the Strategy pattern) at the end of evolution. Crossover is a standard one-point crossover. After the operations, the offspring and mutated chromosomes are always checked for legality, as design patterns may easily be broken. Selection is made with the roulette wheel method.

This approach actually combines the class responsibility assignment problem studied by Simons and Paremee [29,30] and the application of design patterns, as studied by Amoui et al. [32]. Although the selection of design patterns is smaller, the search problem of finding an optimal architecture is much more difficult. First the GA needs to find the optimal class responsibility distribution, and then apply design patterns. In this case the search space grows exponentially, as in order to optimally apply the design patterns, the class responsibility distribution may need to be sub-optimal. This produces a challenge when deciding on the fitness function.

The fitness function is a combination of object-oriented software metrics, most of which are from the Chidamber and Kemerer [45] collection, which have been grouped to measure quality concepts efficiency and modifiability. Some additional metrics have also been developed to measure the effect of communicating through a message dispatcher or interfaces. Furthermore, a complexity measure is introduced. The fitness function is defined as $f = w_1$ PositiveModifiability $-w_2$ NegativeModifiability $+w_3$ PositiveEfficiency $-w_4$ NegativeEfficiency $-w_5$ Complexity, where w_is are weights to be fixed. As discussed, defining the fitness function is the most complex task in all SSBSE problems. In this case, when the problem is so diverse, the fitness function is also intricate: it requires a set of known metrics, a set of special metrics, the grouping of these metrics and additionally weights in order to set preferences to quality aspects.

The approach is tested on a sketch of a medium-sized system [46]. Results show positive development in overall fitness value, while the balancing of weights greatly affects whether the design is more modifiable or efficient. However, the actual designs are not compliant with the fitness values, and would not be accepted by a human architect. This suggests that further improvement is needed in defining the fitness function.

Räihä et al. [34] further develop their work by implementing more design patterns and an alternative approach. In addition to the responsibility dependency graph, a domain model may be given as input. The GA can now be utilized in Model Driven Architecture design, as it takes care of the transformations from Computationally Independent Model to Platform Independent Model. The new design patterns are Mediator and Proxy, and the service oriented architecture style is also implemented by enabling a class to be called through a server. The chromosome representation, mutation and crossover operations, and selection method are kept the same. Results show that the fitness values converge to some optima and reasonable high-level designs are obtained.

In this case the task for the GA is made somewhat easier, as a skeleton of a class structure is given to the algorithm in the form of a domain model. This somewhat eliminates the class responsibility assignment problem and the GA can only concentrate on applying the design patterns. As the results are significantly better, although the search space is more complex when more patterns have been added to the mutations, this suggests that the class responsibility assignment problem is extremely complex on its own, and more research on this would be highly beneficial as a background for several search-based software design related questions.

Räihä et al. [35] continue to develop their approach by including the Template pattern in the design pattern/mutation collection and introducing scenarios as a way to enhance the evaluation of a produced architecture. Scenarios are basically a way to describe an interaction between the system and a stakeholder. In their work, Räihä et al. [35] categorize and formalize modifiability related scenarios so that they can be encoded and given to the GA as an additional part of the fitness function. Each scenario is given a list of preferences regarding the architectural structures that are suitable for that scenario. The preferences are then compared with the suggested architecture and a fitness value is calculated according to how well the given architecture conforms to the preferences. This way the fitness value is more pointed, as the most critical parts of the architecture can be given extra attention and the evaluation is not completely based on general metrics. Results from empirical studies made on two sample systems show that when the scenarios are used, the GA retains the high-speed phase of developing the architecture for 10–20 generations longer than in the case where scenarios are not used. Also, when the scenario fitness is not included in the overall fitness evaluations the GA tends to make decisions that do not support the given scenarios.

Results from this study shows that when the modifications are detailed in applying a design pattern (rather than modifying the architecture "as a whole"), the fitness function also needs to be more focused to study the places in an architecture where such detailed solutions would be most beneficial.

Kessentini et al. [36] also use a search-based approach to model transformations. They start with a small set of examples, from which transformation blocks are extracted, and use particle swarm optimization (PSO) [47]. A model is viewed as a triple of source model, target model and mapping blocks between the source and target models. The source model is formed by a set of constructs. The transformation is only coherent if it does not conflict with the constructs. The transformation quality of a source model (i.e., global quality of a model) is the sum of the transformation qualities of its constructs (i.e., local qualities). This approach is less automated, as the transformations need to be extracted from ready models, and are not general. However, using PSO is especially interesting, and suggests that other algorithms besides GA are also suitable for complex software design problems.

To encode a transformation, an M-dimensional search space is defined, M being the number of constructs. The encoding is now an M-dimensional integer vector whose elements are the mapping blocks selected for each construct. The fitness function is a sum of constructs that can be transformed by the associated blocks multiplied by relative numbers of matched parameters and constructs. The fitness value is normalized by dividing it by $2 * M$, thus resulting in a fitness range of [0, 1].

The method was evaluated and tried with 10 small-size models, of which nine are used as a training set and one as the actual model to be transformed. The precision of model transformation (number of constructs with correct transformations in relation to total number of constructs) is calculated in addition to the fitness values. The best solution was found after only 29 iterations, after which all particles converged to that solution. The test generated 10 transformations. The average precision of these is more than 90%, thus indicating that the transformations would indeed give an optimal result, as the fitness value was also high within the range. The test also showed that some constructs were correctly transformed, although there were no transformation examples available for these particular constructs.

Kim and Park [37] use GAs to dynamically choose components to form the software architecture according to changing demands. The basic concept is to have a set of interchangeable components (e.g., BasicUI and RichUI) which can be selected according to user preferences. The goal is thus to select an optimal architectural instance from all possible instances. This is especially beneficial when the software needs to be transferred, e.g., from a PC to a mobile device.

A softgoal interdependency graph (SIG) is used as a basis for the problem; it represents relationships between quality attributes. The quality attributes are formulated by a set of quality variables. A utility function is used to measure the user's overall satisfaction: the user now gives weights for the quality values to represent their priority. Functional alternatives (i.e., the interchangeable components) are denoted by operationalizing goals. The operationalizing goals can have an impact on a softgoal, i.e., a quality attribute. Alternatives with similar characteristics are grouped by type. One alternative type corresponds to one architectural decision variable. These represent partial configurations of the application. A combination of architectural decision variables comprises an architectural instance.

In addition to the SIG, situation variables and their values are needed as input. Situation variables describe partial information on environmental changes and determine the impacts that architectural decision variables have on the quality attributes. The impact is defined as a situation evaluation function, which is defined for each direct interdependency between an operationalizing goal and quality attribute. Although the fitness function is quite standard, i.e., it calculates the quality through "quality values" and there are weights assigned, the actual computations are not that straightforward. The quality attributes that the fitness function is based on rely on decision variables and situation variables. These in turn need to be calculated by hand, and there is no clear answer as to how the situation variables themselves are gathered.

For the GA, the architectural instance is encoded as a chromosome by using a string of integers representing architectural decisions. Mutation is applied to offspring, for which each digit is subjected to mutation (according to mutation probability). Crossover is a standard two-point crossover. The utility function is used as the fitness function and tournament selection is used for selecting the next generation.

An empirical study is made and compared to exhaustive search. The time needed for the GA is less then $1 * 10^{-5}$ of the time needed for the exhaustive search. The GA also converges to the best solution very quickly, after only 40 generations. Thus, it would seem that using a search algorithm for this

problem would produce extremely good results, at least in terms of time and speed. However, in this case all the components need to be known beforehand, as the task is to choose an optimal set from alternative components. It would be interesting to see at least how all the different variables needed are acquired, and how the approach could be more generalized.

Bodhuin et al. [38] present an approach based on GAs and an environment that, based on previous usage information of an application, re-packages it with the objective of limiting the amount of resources transmitted for using a set of application features. The overall idea is to cluster together (in jars) classes that, for a set of usage scenarios, are likely to be used together. Bodhuin et al. [38] propose to cluster together classes according to dynamic information obtained from executing a series of usage scenarios. The approach aims at grouping in jars classes that are used together during the execution of a scenario, with the purpose of minimizing the overall jar downloading cost, in terms of time in seconds for downloading the application. After having collected the execution trace, the approach determines a preliminary re-packaging considering common class usages and then improves it by using GAs. This approach can be seen as attempting to find optimal sub-architectures for a system, as each jar-package needs to be able to operate on its own. Obviously the success of finding sub-systems greatly depends on how well the class responsibility assignment problem is solved in the system, linking these results to that fundamental problem.

The proposed approach has four steps. First, the application to be analyzed is instrumented, and then it is exercised by executing several scenarios instantiated from use cases. Second, a preliminary solution of the problem is found, grouping together classes used by the same set of scenarios. Third, GAs are used to determine the (sub)-optimal set of jars. Fourth, based on the results of the previous steps, jars are created.

For the GA, an integer array is used as the chromosome representation, where each gene represents a cluster of classes. The initial population is composed randomly. Mutation selects a cluster of classes and randomly changes its allocation to another jar archive. The crossover is the standard one-point crossover. The fitness function is $F(x) = 1/N * \sum(Cost_i)$, where N is the number of scenarios and $Cost$ is calculated from the call cost of making a request to the server and from the class sizes. 10% of the best individuals are kept alive across subsequent generations. Individuals to be reproduced are selected using a roulette-wheel selection. Scenarios are used in a very different way here to in the work of Räihä et al. [35]. Here, scenarios define actions made with the system, and thus contain information of different components of the system that are needed, but do not deal with quality aspects other than how many operations, i.e., scenarios, a certain set of responsibilities is able to perform. Räihä et al. [35], however, use scenarios to describe not functional operations but expectations to the system in terms of quality aspects. These different studies suggest that there are more ways of measuring quality than metrics, and they should be more thoroughly investigated.

Results show that GA does improve the initial packaging, by 60%–90% compared to the actual initial packaging, by 5%–43% compared to a packaging that contains two jars, “used” and “unused”, and by 13%–23% compared to the preliminary best solution. When delay increases, the GA optimization starts to be much more useful than the preliminary optimal solution, while the “used” packaging becomes better. However, for a network delay value lower or slightly higher than the value used for the optimization process, the GA optimization is always the best packaging option. It is found that even when there is a large corpus of classes used in all scenarios, a cost reduction is still possible, even if in such a case the preliminary optimized solution is already a good one. The benefits of the proposed approach depend strongly on several factors, such as the amount of collected dynamic information, the number of scenarios subjected to analysis, the size of the common corpus and the network delay. However, the presented approach and its results can be linked to several other software design related questions, thus raising questions on how different promising results can be combined so that even more complex problems can be solved with search-based methods.

Gold et al. [39] experiment with applying search techniques to integrate the boundary overlapping concept assignment. Hill climbing and GA approaches are investigated. The fixed boundary Hypothesis Based Concept Assignment (HBCA) [48] technique is compared to the new algorithms. As program comprehension is extremely valuable when (re-)designing software architecture, and locating (and understanding) overlapping concepts is one of the most demanding tasks in comprehension, automating this task would significantly save resources in program maintenance.

A concept may take the form of an action or object. For each concept found from source code, a hypothesis is generated and stored. The list of hypotheses is ordered according to the position of the indicators in the source code. The input for the search problem is the hypothesis list. The hypothesis list is given by the application of HBCA. The problem is defined as searching for segments of hypothesis in each hypothesis list according to predetermined fitness criteria such that each segment has the following attributes: each segment contains one or more neighboring hypotheses and there are no duplicate segments.

A chromosome is made up of a set of one or more segment representations, and its length can vary. A segment is encoded as a pair of values (locations) representing the start and end hypothesis of the hypothesis list. All segments with the same winning concept that overlap are compared, and all but the fittest segment are removed from the solution. Tournament selection is used for crossover and mutation. Mutation in GA randomly replaces any hypothesis location within any segment with any other valid hypothesis location with the aim to cause the search to become more randomized. In HC the mutation generates new solutions by selecting a segment and increasing or decreasing one of the values by a single increment. Selecting different mutations for GA and HC is noteworthy: this choice is partially justified by the authors by the fact that mutation is only the secondary operation for the GA, and transformations are primarily done with the crossover. The chosen mutation operator for the

GA seems to ensure diversity within the population. The proposed HC takes advantage of the crossover for GA for the restart mechanism, which recombines all segments to create new pairs of location values, which are then added to the current solution if their inclusion results in an improvement to the fitness value. Crossover utilizes the location of the segments, where only segments of overlapping locations are recombined and the remainder are copied to the new chromosome.

The fitness criteria's aims are finding segments of strongest evidence and binding as many of the hypotheses within the hypothesis list as possible without compromising the segment's strength of evidence. The segmentation strength is a combination of the inner fitness and the potential fitness of each segment. The inner fitness fit_i of a segment is defined as $signal_i$–$noise_i$, where $signal_i$ is the number of hypotheses within the segment that contribute to the winner, and $noise_i$ represents the number of hypotheses within the segment that do not contribute to the winner. In addition, each segment is evaluated with respect to the entire segment hypothesis list: the potential segment fitness, fit_p, is evaluated by taking account of $signal_p$, the number of hypotheses outside of the segment that could contribute to the segment's winning concept if they were included in the segment. The potential segment fitness is thus defined as $fit_p = signal_i - signal_p$. The overall segment fitness is defined as $segfit = fit_i + fit_p$. The total segment fitness is a sum of segment fitnesses. The fitness is normalized with respect to the length of the hypothesis list. The chosen fitness function seems quite simple when broken down to actual calculations. This further confirms the findings, by e.g. [42], that simple approaches tend to have promising results, as there is less room to err.

An empirical study is used. Results are also compared to sets of randomly generated solutions for each hypothesis list, created according to the solutions structure. The results from GA, HC and random experiment are compared based on their fitness values. The GA fitness distribution is the same as those of HC and random, but achieves higher values. HC is clearly inferior. Comparing GA, HC and HBCA shows a lack of solutions with low Signal to Noise ratios for GA and HC when compared to HBCA. GA is identified as the best of the proposed algorithms for concept assignment which allow overlapping concept boundaries. Also, the HC results are somewhat disappointing as they are found to be significantly worse than GA and random solutions. However, HC produces stronger results than HBCA on the signal to size measure. The GA and HC are found to consistently produce stronger concepts than HBCA. It might be worth studying how the HC would have performed if it used the same mutation operator as the GA. Although the GA primarily used the crossover, which was used as a basis for the HC, the GAs large population makes the application of this operator significantly more different than with HC.

Goldsby and Chang [40] and Goldsby et al. [41] study the digital evolution of behavioral models for autonomic systems with Avida. It is difficult to predict the behavior of autonomic systems before deployment, and thus automatic generation of behavioral models greatly eases the task of software engineers attempting the comprehend the system. In digital evolution a population of self-replicating computer programs (digital organisms) exists in a computational environment and is subject to mutations and selection. In this approach each digital organism is considered as a generator for a UML state diagram describing the systems behavior.

Each organism is given instinctual knowledge of the system in the form of a UML class diagram representing the system structure, as well as optional seed state diagrams. A genome is thus seen as a set of instructions telling the system how to behave. The genome is also capable of replicating itself. In fact, at the beginning of each population there exists only one organism that only knows how to replicate itself, thus creating the rest of the population. Mutations include replacing an instruction, inserting an additional instruction or removing an instruction from the genome. As genomes are self-replicating, crossover is not used in order to create offspring. Here the choice of UML state diagrams is clever, as it visualizes the behavior in quite a simple manner, making the interpretation of the result easy. Also the choice of encoding conforms well to the chosen visualization method. However, the actual encoding of rules into the genome is not simple, and requires several different alphabets and lists of variables.

The fitness or quality of an organism is evaluated by a set of tasks, defined by the developer. Each task that the behavioral model is able to execute increases its merit. The higher a merit an organism has, the more it will replicate itself, eventually ending up dominating the population. This is yet another instance where the fitness is measured by something other than traditional metrics.

A behavioral model of an intelligent robot is used as a case study for Avida. Through 100 runs of Avida, seven behavioral models are generated for the example system. Post-evolution analysis includes evaluation with the following criteria: minimum states, minimum transitions, fault tolerance, readability and tolerance. After the analysis, one of the models meets all but one criterion (safety) and three models meet three of the five criteria. One model does not meet any of the additional criteria. Thus, the produced behavioral models would seem to be of average quality.

Lutz [42] uses a measure based on an information theoretic minimum description length principle [49] to compare hierarchical decompositions. This measure is furthermore used as the fitness function for the GA which explores the space of possible hierarchical decompositions of a system. Although this is very similar to software clustering, this approach is considered as architecture design as it does not need an initial clustering to improve, but designs the clustering purely based on the underlying system and its dependencies.

In hierarchical graphs, links can represent such things as dependency relationships between the components of control-flow or data-flow. In order to consider the best way to hierarchically break a system up into components, one needs to know what makes a hierarchical modular decomposition (HMD) of a system better than another. Lutz takes the view that the best HMD of a system is the simplest. In practice this seems to give rise to HMDs in which modules are highly connected internally (high cohesion) and have relatively few connections which cross module boundaries (low coupling), and thus seems to achieve a principled trade-off between the

coupling and cohesion heuristics without actually involving either. This also suggests that high quality architectures can effectively be identified through subjective inspection. A human architect may quite easily say if one design appears simpler than another, while calculation cohesion and coupling values are more time consuming and complex.

For the GA, the genome is a HMD for the underlying system. The chromosomes in the initial population are created by randomly mutating some number of times a particular "seed" individual. The initial seed individual is constructed by modularizing the initial system. Three different mutation operations are used that can all be thought of as operations on the module tree for the HMD. They are: 1. moving a randomly chosen node from where it is in the tree into another randomly chosen module of the tree, 2. modularizing the nodes of some randomly chosen module, i.e., creating a new module containing the basic entities of some module, and 3. removing a module "boundary". The crossover operator resembles the tree-based crossover operation used in genetic programming and is most easily considered as a concatenating operation on the module trees of the two HMDs involved. However, legal solutions are not guaranteed, and illegal ones are repaired.

The tree-like structure is significantly more complex than usual genome encodings for a GA. This is of course in line with the demands of the problem of finding an optimal HMD, but also reflects on the understandability of the chosen operations. The operations are difficult (if not impossible) to completely understand without visualization, and difficult corrective operations are needed in order to keep the system structure intact. The analogy between the chosen tree-operations and actual effects to the architecture is also quite difficult to grasp.

The fitness is given as 1/complexity. Among other systems, a real software design is used for testing. A HMD with significantly lower complexity than the original was found very reliably, and the system could group the various components of the system into a HMD exhibiting a very logical (in terms of function) structure. These results validate that using simplicity as a fitness function is justified.

3.1.3. *Summarizing remarks*

Search-based approaches to software architecture design is clearly a diverse field, as the studies presented solve very different issues relating to OO software architecture design and program comprehension. Some consensus can be found in the very basics: solving the class responsibility assignment problem, applying design choices to create an architecture and finding an optimal modularization (Lutz [42] creates a modularization, Kim and Park [37] attempt to find an optimal set of components and Bodhuin et al. [38] attempt to find optimal sub-architectures). However, even within these sub-areas of OO design, the approaches are quite different, and practically no agreement can be found when studying the chosen encodings, operations or fitness function. What is noticeable, however, is that several approaches to quite different problems within this area use a fitness function that is not based on metrics. This highlights the need for better validation when using metrics in evaluating the quality of software, and especially software architectures. Many metrics need source code and very detailed information; this alone suggests that they are not suitable for this higher level problem.

3.2. *Service-oriented architecture design*

3.2.1. *Background*

Web services are rapidly changing the landscape of software engineering, and service-oriented architectures (SOA) are especially popular in business. One of the most interesting challenges introduced by web services is represented by Quality of Service (QoS)-aware composition and late-binding. This allows binding, at run-time, a service-oriented system with a set of services that, among those providing the required features, meet some non-functional constraints, and optimize criteria such as the overall cost or response time. Hence, QoS-aware composition can be modeled as an optimization problem. This problem is NP-hard, which makes it suitable for meta-heuristic search algorithms. For more background on SOA, see, e.g., [50]. Section 3.2.2 describes several approaches that have used a GA to deal with optimizing service compositions. Summarizing remarks on the different approaches are given in Section 3.2.3, and the fundamentals of each approach are collected in Table 3.

3.2.2. *Approaches*

Canfora et al. [51] propose a GA to optimize service compositions. The approach attempts to quickly determine a set of concrete services to be bound to the abstract services composing the workflow of a composite service. Such a set needs both to meet QoS constraints, established in the Service Level Agreement (SLA), and to optimize a function of some other QoS parameters.

A composite service S is considered as a set of n abstract services $\{s_1, s_2, \ldots, s_n\}$, whose structure is defined through some workflow description language. Each component s_j can be bound to one of the m concrete services, which are functionally equivalent. Computing the QoS of a composite service is made by combining calculations for quality attributes *time*, *cost*, *availability*, *reliability* and *customer attraction*. Calculations take into account Switch, Sequence, Flow and Loop patterns in the workflow.

The genome is encoded as an integer array whose number of items equals the number of distinct abstract services composing the services. Each item, in turn, contains an index to the array of the concrete services matching that abstract service. The mutation operator randomly replaces an abstract service with another one among those available, while the crossover operator is the standard two-point crossover. This can be seen as an attempt to preserve building blocks, i.e., sequences of optimal service bindings. Abstract services for which only one concrete service is available are taken out of the GA evolution.

The fitness function needs to maximize some QoS attributes, while minimizing others. In addition, the fitness function must penalize individuals that do not meet the constraints and drive the evolution towards constraint satisfaction, the distance from which is denoted by D. The fitness function is $f = (w_1\text{Cost} + w_2\text{Time})/(w_3\text{Availability} +$

Table 3 – Studies in search-based service-oriented software architecture design.

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Canfora et al. [51]	Service composition with respect to QoS attributes	Sets of abstract and concrete services	Integer array, whose size is the number of abstract services, each item contains an index to array of concrete services	Randomly replaces an abstract service with another	Standard two-point crossover	Minimize cost and time, maximize availability and reliability, meet constraints, with penalty	Optimized service composition meeting constraints, concrete services bound to abstract services	A dynamic penalty was experimented with
Canfora et al. [52]	Replanning during execution time	Sets of abstract and concrete services	Integer array, whose size is the number of abstract services, each item contains an index to array of concrete services	Randomly replaces an abstract service with another	Standard two-point crossover	Minimize cost and time, maximize availability and reliability, meet constraints	Optimized service composition meeting constraints, concrete services bound to abstract services	GA used to calculate initial QoS-value and QoS-values inbetween: replanning is triggered by other algorithms
Jaeger and Mühl [53]	Service assignment with respect to QoS attributes	Selection of services and tasks to be carried out	A tuple representing an assignment of a candidate for a task	Changes an individual task-candidate assignment	Combining task-candidate assignments	Minimize cost and time, maximize availability and reliability, meet constraints, with penalty	Tasks assigned to services considering QoS attributes	A trade-off couple between execution time and cost is defined
Zhang et al. [54]	Task assignment with relation to QoS attributes	Selections of tasks and services	Relation matrix coding scheme	Standard, with corrective function	Standard, with corrective function	Minimize cost and time, maximize availability and reliability, meet constraints	Tasks assigned to services considering QoS attributes	Initial population and mutation policies defined
Su et al. [55]	Task assignment with relation to QoS attributes	Selections of tasks and services	Relation matrix coding scheme	Standard, with corrective function	Standard, with corrective function	Minimize cost and time, maximize availability and reliability, meet constraints	Tasks assigned to services considering QoS attributes	Initial population and mutation policies defined
Cao et al. [56,57]	Business process optimization	Collections of web services and service agents (SAg) composing a business process	Integer encoding, assigning a SAg to a service	Changes the service to which a SAg is bound with corrective function	Standard one-point, producing two new offspring with corrective function	Cost	Services assigned to service agents	

w_4Reliability) + w_5D. QoS attributes are normalized in the interval [0, 1). Although the fitness function seems simple in this way, the actual calculations behind the different attributes are complex. The values are achieved by calculating the quality value for each attribute for each pattern in the workflow. The actual functions to define how these values are calculated are not defined, and it would be interesting to see how, e.g., availability is achieved, as this would show the amount of information needed as input to calculate the fitness value. The weights $w_1, \ldots, w_5$ are positive reals. Normalizing the fitness evaluators ensures that the weights have the true effect to the fitness value that they are meant to have.

A dynamic penalty is experimented with, so that w_5 is increased over the generations. An elitist GA is used, where the best two individuals are kept alive across generations. The roulette wheel method is used for selection.

The GA is able to find solutions that meet the constraints, and optimizes different parameters (here cost and time). Results show that the dynamic fitness does not outperform the static fitness. Even different calibrations of weights do not help. The convergence times of GA and Integer Programming (IP) [58] are compared for the (almost) same achieved solution. The results show that when the number of concrete services is small, IP outperforms GA. For about 17 concrete services, the performance is about the same. After that, GA clearly outperforms IP. Thus, as SOA is most useful when the number of services is large, it would seem that GA is a worthwhile solution to optimizing the service-binding.

Canfora et al. [52] have continued their work by using a GA in replanning the binding between a composite service and its invoked services during execution. Replanning is triggered once it can be predicted that the actual service QoS will differ from initial estimates. After this, the slice, i.e., the part of workflow still remaining to be executed, is determined and replanned. The used GA approach is the same as earlier, but additional algorithms are used to trigger replanning and computing workflow slices. The GA is used to calculate the initial QoS-values as well as optimizing the replanned slices. Experiments were made with realistic examples, and results concentrate on the cost quality factor. The algorithms managed to reduce the final cost from the initial estimate, while response time increased in all cases. The authors end with a note that the trade-off between response time and cost quality factors needs to be examined thoroughly in the future.

Jaeger and Mühl [53] discuss the optimization problem when selecting services while considering different QoS characteristics. A GA is implemented and tested on a simulation environment in order to compare its performance with other approaches.

An individual in the implemented GA represents an assignment of a candidate for each task and can be represented by a tuple. A population represents a set of task-candidate assignments. The initial population is generated arbitrarily from possible combinations of tasks and candidates. Mutation changes a particular task-candidate assignment of an individual. Crossover is made by combining two particular task-candidate assignments to form new ones and depends on the fitness value. The fitness value is computed based on the QoS resulting from the encoded task-services assignment. Jaeger and Mühl use the same fitness function as Canfora et al. [51,52] in order to get comparable results.

A trade-off couple between execution time and cost is defined as follows: the percentage a, added to the optimal execution time, is taken to calculate the percentage b, added to the optimal cost, with $a + b = 100$. Thus, the shorter the execution time is, the worse will be the cost and vice versa. The constraint is determined to perform the constraint selection on the execution time first. The aggregated cost for the composition is increased by 20% and then taken as the constraint that has to be met by the selection. This appears as an attempt to answer the problem noted by Canfora et al. [52] in their later study.

Several variations of the fitness function are possible. Jaeger and Mühl [53] use a multiplication of the fitness to make the difference between weak and strong fitnesses larger. When the multiplying factor is 4, it achieves higher QoS values than those with a smaller factor; however, a factor of 8 does not achieve values as high. The scaled algorithm performed slightly better than the one with a factor of 2, and behaved similarly to the weighted algorithm. The penalty factor was also investigated, and it was varied between 0.01 and 0.99 in steps of 0.01. The results show that a factor of 0.5 would result in few cases where the algorithm does not find a constraint meeting solution. On the other hand, solutions below 0.1 appear too strong, as they represent an unnecessary restriction of the GA to evolve further invalid solutions. These different experiments on some very basic parameters demonstrate the difficulty of optimizing the GA: even the more simple choices are anything but straightforward.

The GA offers a good performance at feasible computational efforts when compared to, e.g., bottom-up heuristics. However, this approach shows a large gap when compared to the resulting optimization of a branch-and-bound approach or to exhaustive search. It appears that the considered setup of values along with the given optimization goals and constraints prevent a GA from efficiently identifying very near optimal solutions.

Zhang et al. [54] implement a GA that, by running only once, can construct the composite service plan according to the QoS requirements from many services compositions. This GA includes a special relation matrix coding scheme (RMCS) of chromosomes proposed on the basis of the characters of web services selection.

By means of the particular definition, it can simultaneously represent all paths of services selection. Furthermore, the selected coding scheme can simultaneously denote many web service scenarios that the one dimension coding scheme can not express at one time.

According to the characteristic of the services composition, the RMCS is adopted using a neighboring matrix. In the matrix, n is the number of all tasks included in services composition. The elements along the main diagonal for the matrix express all the abstract service nodes one by one, and are arranged from the node with the smallest code number to the node with the largest code number. The objects of the evolution operators are all elements along the main diagonal of the

matrix. The chromosome is made up of these elements. The other elements in the matrix are to be used to check whether the created new chromosomes by the crossover and mutation operators are available and to calculate the QoS values of chromosomes. This appears to mainly combine the integer array and the table of services linked to it, used by Canfora et al. [51], into one data structure. The tuple representation chosen by Jaeger and Mühl [53] does not seem that different either, as a tuple can basically contain the information of what is represented by a column and a row in a matrix.

The policy for initial population attempts to confirm the proportion of chromosomes for every path to the size of the population. The method is to calculate the proportion of compositions of every path to the sum of all compositions of all paths. The more compositions there are of one path, the more chromosomes for the path there are in the population.

The value of every task in every chromosome is confirmed according to a local optimized method. The larger the value of QoS of a concrete service is, the larger the probability to be selected for the task is. Roulette wheel selection is used to select concrete services for every task.

The probability of mutation is for the chromosome instead of the locus. If mutation occurs, the object path will be confirmed firstly as to whether it is the same as the current path expressed by the current chromosome. If the paths are different, the object path will be selected from all available paths except the current one. If the object is itself, the new chromosome will be checked to see whether it is the same as the old chromosome. The same chromosome will result in the mutation operation again. If the objects are different paths from the current path, a new chromosome will be related on the basis of the object path.

A check operation is used after the invocations of crossover and mutation. If the values of the crossover loci in two crossover chromosomes are all for the selected web services, the new chromosomes are valid. Else, the new chromosomes need to be checked on the basis of the relation matrix. Mutation checks are needed if changed from a selected web service to a certain value or vice versa.

Zhang et al. [54] compared the GA with RMCS to a standard GA with the same data, including workflows of different sizes. The used fitness function is as defined by Canfora et al. [59]. The coding scheme, the initial population policy and the mutation policy are the differences between the two GAs. Results show that the novel GA outperforms the standard one in terms of achieved fitness values. As the number of tasks grows, so does the difference in fitness values (and performance time, in favor of the standard solution) between the two GAs. The weaknesses of this approach are thus long running time and slow convergence. Tests on the initial population and the mutation policies show that as the number of tasks grows, the GA with RMCS more clearly outperforms the standard one. Thus it would seem that combining the information into a heavier data structure, a matrix, increases execution time significantly. Also, as it is noted that the improvement fitness values with the novel GA for larger task sets is achieved by testing other improvement than the encoding, the true achievements are the ones that really differ from previous approaches, rather than the new representation. Tests on the coding scheme show that the novel matrix approach only achieves noticeably better fitness values when the number of tasks is increased (although the improvement is not linear): the fitness values for 10 tasks differ by less than 1%, the fitness values for 25 tasks differ by approximately 30%, and the fitness values for 30 tasks by approximately 20%. Another interesting point is the choice of parameters: Zhang et al. [54] use 10 000 generations and 400 individuals for a population in their tests. However, the standard GA seems to achieve its optimum after 1000 generations and the one with the novel encoding after 3000 generations. Thus one wonders about the need for such unusual parameter selections.

Zhang et al. [54] report that experiments on QoS-aware web services selection show that the GA with the presented matrix approach can get a significantly better composite service plan than the GA with the one dimensional coding scheme, and that the QoS policies play an important role in the improvement of the fitness of the GA.

Su et al. [55] continue the work of Zhang et al. [54] by proposing improvements for the fitness function and mutation policy. An objective fitness function 1 (OF1) is first defined as a sum of quality factors and weights, providing the user with a way to show favoritism between quality factors. The sum of positive quality factors is divided by the sum of negative quality factors. The second fitness function (OF2) is a proportional one and takes into account the different ranges of quality value. The third fitness function (OF3) combines OF1 and OF2, producing a proportional fitness function that also expresses the differences between negative and positive quality factors. Thus Su et al. seem to have noticed the problems with defining the fitness functions, as the fitness function actually used by Canfora et al. [51,52] includes similar improvements.

Four different mutation policies are also inspected. Mutation policy 1 (MP1) operates so that the probability of the mutation is tied to each locus of a chromosome. Mutation policy 2 (MP2) has the mutation probability tied to the chromosomes. Mutation policy 3 (MP3) has the same principle as MP1, except that now the child may be identical to the parent. Mutation policy 4 (MP4) has the probability tied to each locus, and has an equal selection probability for each concrete service and the "0" service.

Experiments with the different fitness functions suggest that OF3 clearly outperforms OF1 and OF2 in terms of the reached average maximum fitness value. This is quite unsurprising, as OF3 is the most developed fitness function. Experiments on the different mutation policies show that MP1 obtains the best fitness values while MP4 performs the worst.

Cao et al. [56,57] present a service selection model using GA to optimize a business process composed of many service agents (SAg). Each SAg corresponds to a collection of available web services provided by multiple-service providers to perform a specific function. Service selection is an optimization process taking into account the relationships among the services. When only measuring cost, the service selection is equivalent to a single-objective optimization problem. Better performance is achieved using GA compared to using a local service selection strategy.

An individual is generated for the initial population by randomly selecting a web service for each SAg of the services flow, and the newly generated individual is immediately checked as to whether the corresponding solution satisfies the constraints. If any of the constraints is violated, then the generated individual is regarded as invalid and discarded. Roulette wheel selection is used for individuals to breed.

Mutation bounds the selected SAg to a different web service than the original one. After an offspring is mutated, it is also immediately checked whether the corresponding solution is valid. If any constraints are violated, then the mutated offspring is discarded and the mutation operation is retried.

A traditional single-point crossover operator is used to produce two new offspring. After each crossover operation, the offspring are immediately checked as to whether the corresponding solutions are valid. If any of the constraints is violated, then both offspring are discarded and the crossover operation for the mated parents is retried. If valid offspring still cannot be obtained after a certain number of retries, the crossover operation for these two parents is given up to avoid a possible infinite loop.

Cao et al. [56,57] take cost as the primary concern of many business processes. The overall cost of each execution path can always be represented by the summation cost of its subset components. For GA, integer encoding is used. The solution to service selection is encoded into a vector of integers. The fitness function is defined as $f = U - \sum$ (costs of service flows), if cost $< U$, and otherwise 0. The constant U should be selected as an appropriate positive number to ensure all good individuals get a positive fitness value in the feasible solution space. On the other hand, U can also be utilized to adjust the selection pressure of GA. This is a clever approach to give the developer a simple way to adjust the selection process and appreciation of different solutions.

In the case study the best fitness of the population has a rapid increase at the beginning of the evolution process and then convergences slowly. It means the overall cost of the SAg is generally decreasing with the evolution process. For better solutions, the whole optimization process can be repeated for a number of times, and the best one in all final solutions is selected as the ultimate solution to the service selection problem.

3.2.3. *Summarizing remarks*

Contrary to the studies relating to OO architecture design, the approaches to apply search algorithms in SOA design are extremely similar. Nearly all studies use the same fitness functions or they have made only small modifications to it. Also the basic representation of the problem is very similar; although different definitions are used, the underlying problem is always linking concrete services with abstract services. Improvements have been attempted by creating different initial population and mutation policies; note, that the actual mutation is still the same, but the way the mutation is applied is changed. Additionally, there is no consensus in the encoding of the solution, although the problem is the same, and some tests have been made to compare different encoding options. Thus the main questions in this area seem to be: are there other problems in SOA where search algorithms could be applied, and can a truly optimal encoding be found to the currently studied problem? Additionally, the fitness function deserves much more attention and testing, as the developers of the fitness function used by all the studies say themselves that the relationships and trade-offs between different quality attributes need to be carefully studied. Results with dynamic fitness functions also interestingly did not increase the fitness value. Räihä et al. [33,34] experimented with dynamic mutations, but discarded them in their latest study [35]. This would suggest that using dynamicity with GAs is a complex problem, demanding well-defined operations and firm justifications for the use of such improvements before adding them to the experiments.

3.3. *Other*

3.3.1. *Background*

In addition to purely designing software architecture, there are some factors that should be optimized, regardless of the particularities of an architecture. Firstly, there is the reliability-cost tradeoff. The reliability of software is always dependent on its architecture, and the different components should be as reliable as possible. However, the more work put in to ensure reliability of different components, the more the software will cost. Wadekar and Gokhale [60] implement a GA to optimize the reliability-cost tradeoff. Secondly, there are some parameters, e.g., tile sizes in loop tiling and loop unrolling, which can be optimized for all software architectures in order to optimize the performance of the software. Che et al. [61] apply search-based techniques for such parameter optimization.

3.3.2. *Approaches*

Wadekar and Gokhale [60] present an optimization framework founded on architecture-based analysis techniques, and describe how the framework can be used to evaluate cost and reliability tradeoffs using a GA. The methodology for the reliability analysis of a terminating application is based on its architecture. The architecture is described using the one-step transition probability matrix P of a discrete time Markov chain (DTMC).

Wadekar and Gokhale assume that the reliabilities of the individual modules are known, with R_i denoting the reliability of module i. It is also assumed that the cost of the software consisting of n components, denoted by C, can be given by a generic expression of the form: $C = C_1(R_1) + C_2(R_2) + \cdots + C_n(R_n)$, where C_i is the cost of component i and depends monotonically on the reliability R_i. Thus, the problem of minimizing the software cost while achieving the desired reliability is the problem of selecting module reliabilities.

A chromosome is a list of module reliabilities. Each member in the list, a gene, corresponds to a module in the software. The independent value in each gene is the reliability of the module it represents, and the dependent value is the module cost given by the module cost-reliability relation or a table known *a priori*. The gene values are changed to alter the cost and reliability of a software implementation represented by a particular chromosome.

Mutation and crossover operations are standard. To avoid convergence to a local optimum as the population size increases, the mutation operation is used more frequently. A cumulative-probability based basic selection mechanism is used for selection. Chromosomes are ranked by fitness and divided into rank groups. The probability of selection of chromosomes varies uniformly according to their rank group, where chromosomes in the first rank group have the largest probability. A new generation of the population is created by selecting $p_{imax}/2$ chromosomes, where p_{imax} is maximum population. If the cost reduction is less than or equal to φ% of the current best cost τ number of times, the GA terminates. During any generation cycle if the cost reduction is larger, the counter τ is reset to 0. The reduction percentage factor φ and the counter limit τ are parameters. This approach is one of the few alternatives used to terminate a GA, as most studies presented use a straightforward generation number to terminate the execution of the algorithm.

The fitness function is $f = (-K/lnR)/C^{\gamma}$, where K is a large positive constant. The fitness of solutions increases superlinearly with their reliability. The constant γ is used to linearize the cost variation. The maximum fitness is directly proportional to K. An intermediate value of gamma, $\gamma = 1.5$, allows the GA to distinguish between low-cost and high-cost solutions, while selecting a sufficient number of high-cost high-reliability solutions that may generate the optimal high-reliability low-cost solution.

Wadekar and Gokhale [60] compare the GA against exhaustive search. The results indicate that the GA consistently and efficiently provides optimal or very close to optimal designs, even though the percentage of such designs in the overall feasible design space is extremely small. The results also highlight the robustness of the GA. However, the small number of near-optimal solutions demonstrates that the fitness landscape is very complex, again conforming to the need to extensively investigate the cost-reliability trade-off. The case study results show how the GA can be effectively used to select components such that the software cost is minimized, for various cost structures.

Che et al. [61] present a framework for performance optimization parameter selection, where the problem is transformed into a combinatorial minimization problem. Many performance optimization methods depend on the right optimization parameters to get good performance for an application. Che et al. search for the near optimal optimization parameters in a manner that is adaptable for different architectures. First, a reduction transformation is performed to reduce the program's runtime while maintaining its relative performance with regard to different parameter vectors. The near-optimal optimization parameter vector based on the reduced program's real execution time is searched by GA, which converges to a near-optimal solution quickly. The reduction transformation reduces the time to evaluate the quality of each parameter vector.

First some transformations are applied to the application, leaving the optimization parameter vector to be read from a configuration file. Second, the application is compiled into an executable with the native compiler. Then the framework repeatedly generates the configure file with a different parameter vector selected by search and then measures the executable's runtime.

The chromosome encoding for the GA is a vector of integer values, with each integer corresponding to an optimization parameter of a solution. No illegal solutions are allowed. The population has a fixed size. A simple integer value mutation is implemented and an integer number recombination scheme is used for crossover. The fitness value reflects the duality of an individual in relation to other individuals. The linear rank-based fitness assignment scheme is used to calculate the fitness values. Selection for a new generation is made by elitism and the roulette wheel method. Test results show that the GA can adapt to different execution environments automatically. For each platform, it always selects excellent optimization parameters for 80% of programs. Results show that the number of individuals evaluated is far smaller than the size of the solution space for each program on each platform. The optimization time is also small.

4. Software clustering

4.1. *Background*

As software systems develop and are maintained, they tend to grow in size and complexity. A particular problem is the growing number of dependencies between libraries, modules and components within the modules. Software clustering (or modularization) attempts to optimize the clustering of components into modules in such a way that there are as many dependencies within a module as possible and as few dependencies between modules as possible. This will enhance the understandability of a system, which in turn will make it more maintainable and modifiable. Also, fewer dependencies between modules usually results in better efficiency.

As components or modules (depending on the level of detail in the chosen representation) can be depicted as vertices and dependencies between them as edges in a graph, the software clustering problem can be traced back to a graph partitioning problem, which is NP-complete. Genetic algorithms have successfully been applied to a general graph partitioning problem [62,63], and thus, the related software clustering problem is most suitable for meta-heuristic search techniques.

Although the basic problem is relatively simple to define and the goodness of a modularization can be calculated based on the goodness of the underlying graph partitioning, the nature of software systems provides challenges when defining the actual fitness function for the optimization algorithm. Also, not all necessary information can be encoded into a simple graph representation, and this presents another question to be answered when designing a search-based approach for modularization. Section 4.2 presents approaches using GAs, HC and SA to find good software modularizations, after which summarizing remarks are presented in Section 4.3 and the fundamentals of each study are collected in Table 4.

Table 4 – Research approaches in search-based software clustering.

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Mancoridis et al. [64]	Automation of partitioning components of a system into clusters	System given as a module dependency graph (MDG)	MDG	N/A	N/A	Minimize inter-connectivity, maximize intra-connectivity, combined as modularization quality (MQ)	Optimized clustering of system	
Doval et al. [65]	Automation of partitioning components of a system into clusters	MDG	String of integers	Standard	Standard	MQ	Optimized clustering of system	Continued work from Mancoridis et al. [64] by implementing a GA
Mancoridis et al. [66]	Automation of partitioning components of a system into clusters	MDG	MDG	N/A	N/A	MQ	Optimized clustering of system	Continued work from Mancoridis et al. [64]; characteristics of modules taken into account in clustering operations
Mitchell and Mancoridis [67–69]	Automation of partitioning components of a system into clusters	MDG	String of integers	Standard	Standard	MQ as a sum of clustering factors	Optimized clustering of system	Continued work from Doval et al. [65]; new definition of the modularization quality and an enhanced HC algorithm
Mitchell and Mancoridis [69,70]	Automation of partitioning components of a system into clusters	MDG	String of integers	Standard	Standard	MQ, search landscape	Optimized clustering of system	Continued work from Mitchell and Mancoridis [67–69]; search landscape taken into account
Mitchell et al. [71]	Automated reverse engineering from source code to architecture	Source code of application	N/A	N/A	N/A	Quality based on use and style relations	Software architecture	HC and edge removal are used as search algorithms from MDG to architecture

(continued on next page)

Table 4 (continued)

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Mahdavi et al. [72,73]	Automated clustering of system	MDG	String of integers	Standard	Standard	MQ	Optimized clustering of system	Multiple hill climbs are used as search algorithm; building blocks are preserved by using parallel hill climbs
Harman et al. [74]	New encoding and crossover introduced	System as modules and elements	Look-up table for modules	Move component from one module to another	New crossover, preserves partial module allocations	Maximize cohesion, minimize coupling	Optimized clustering	
Harman et al. [75]	Comparison of robustness between two fitness functions	Clustered system	N/A	N/A	N/A	MQ compared against EVM	-	
Antoniol et al. [76]	Cluster optimization	System containing applications and libraries	Bit matrix	Two random rows of a column in matrix are swapped or an object is cloned by changing a value from zero to one	A random column is taken as split point and contents are swapped	Inter-library dependencies, number of object-application links and size of libraries	Optimized clustering, sizes and dependencies between libraries diminished	Optimal number of clusters is calculated for a matrix with the Silhouette statistic
Di Penta et al. [77]	A refactoring framework taking into account several aspects of software quality when refactoring existing system.	Software system as a system graph SG	Bit matrix; each library of clusters is represented by a matrix	Swapping two bits in a column or changing a value from 0 to 1 (taking into account preconditions)	N/A	Dependency factor, partitioning ratio, standard deviation and feedback	Refactored libraries	HC and GA used.
Hyunh and Cai [78]	Conformance check of actual design to suggested design	Design structure matrices for design and source code (DSM)	Graph constructed of DSM	N/A	N/A	Graph edit distance, penalty and differentiation between graphs with same distance	Optimized clustering of actual design conforming to suggested design	

4.2. Approaches

Mancoridis et al. [64] treat automatic modularization as an optimization problem and have created the Bunch tool that uses HC and GA to aid its clustering algorithms. A hierarchical view of the system organization is created based solely on the components and relationships that exist in the source code. The first step is to represent the system modules and the module-level relationships as a module dependency graph (MDG). An algorithm is then used to partition the graph in a way that derives the high-level subsystem structure from the component-level relationships that are extracted from the source code. The goal of this software modularization process is to automatically partition the components of a system into clusters (subsystems) so that the resultant organization concurrently minimizes inter-connectivity while maximizing intra-connectivity. This task is accomplished by treating clustering as an optimization problem, where the goal is to maximize an objective function based on a formal characterization of the trade-off between inter- and intra-connectivity. Intuitively, intra-connectivity could be seen as cohesion and inter-connectivity as coupling.

The clusters, once discovered, represent higher-level component abstractions of a system's organization. Each subsystem contains a collection of modules that either cooperate to perform some high-level function in the overall system or provide a set of related services that are used throughout the system. Intra-connectivity A_i of cluster i consisting of N_i components and m_i intra-edge dependencies as $A_i = m_i/N_i^2$, bound between 0 and 1. Inter-connectivity measures the connectivity between two distinct clusters. A high degree of inter-connectivity is an indication of poor subsystem partitioning. Inter-connectivity E_{ij} between clusters i and j consisting of N_i and N_j components with e_{ij} inter-edge dependencies is 0, if $i = j$, and $e_{ij}/2 * N_i N_j$ otherwise, bound between 0 and 1. Modularization Quality (MQ) demonstrates the trade-off between inter- and intra-connectivities, and it is defined for a module dependency graph partitioned into k clusters as $1/k * \sum \frac{A_i - 1}{k * \frac{k-1}{2}} * \sum E_{i,j}$ if $k > 1$, or $A_{1,}$ if $k = 1$.

The first step in automatic modularization is to parse the source code and build a MDG. A sub-optimal clustering algorithm works as the traditional hill climbing one by randomly selecting a better neighbor. The GA starts with a population of randomly generated initial partitions and systematically improves them until all of the initial samples converge. The GA uses the "neighboring partition" definition to improve an individual, and thus only contains one mutation operator, which is the same one as used with HC. Selection is done by randomly selecting a percentage of N partitions and improving each one by finding a better neighboring partition. A new population is generated by making N selections, with replacements for the existing population of N partitions. Selections are random and biased in favor of partitions with larger MQs. The algorithm continues until no improvement is seen for t generations, or until all of the partitions in the population have converged to their maximum MQ, or until the maximum number of generations has been reached. The partition with the largest MQ in the last population is the sub-optimal solution.

Experimentation with this clustering technique has shown good results for many of the systems that have been investigated. The primary method used to evaluate the results is to present an automatically generated modularization of a software system to the actual system designer and ask for feedback on the quality of the results. A case study was made and the results were shown to an expert, who highly appreciated the result produced by Bunch.

The validation of the method is interesting, as the original designer of a system should be the one who knows the system best, and thus should be the best one to evaluate designs of the system. It is also encouraging that the designers were open and admitted that the tool was able to improve the design that they must have though of as optimal at some point. This indicates that there truly is a place for software design tools if the methods are well-defined enough.

Doval et al. [65] have implemented a more refined GA in the Bunch tool, as it now contains a crossover operator and more defined mutation and crossover rates. The effectiveness of the technique is demonstrated by applying it to a medium-sized software system. For encoding, each node in the graph (MDG) has a unique numerical identifier assigned to it. These unique identifiers define which position in the encoded string will be used to define that node's cluster. Mutation and crossover operators are standard. A roulette wheel selection is used for the GA, complemented with elitism. The fitness function is based on the MQ metric.The crossover rate was 80% for populations of 100 individuals or fewer and 100% for populations of a thousand individuals or more, varying linearly between those values. The mutation rate is 0.004 $\log_2(N)$. The MQ values for constant population and generation values were smaller, but fairly close, within 10% of the final values achieved for population and generation.

The affect of the population size to crossover rate is interesting, especially in the sense that with smaller populations the rate is smaller. Intuitively it would seem that with larger populations there would be a higher chance that the population contains some extremely poor individuals, the parts of which are not worthwhile to pass on to future generations.

Mancoridis et al. [66] have continued to develop the Bunch tool for optimizing modularization. Firstly, almost every system has a few modules that do not seem to belong to any particular subsystem, but rather, to several subsystems. These modules are called omnipresent, because they either use or are used by a large number of modules in the system. In the improved version users are allowed to specify two lists of omnipresent modules, one for clients and another for suppliers. The omnipresent clients and suppliers are assigned to two separate subsystems.

Secondly, experienced developers tend to have good intuition about which modules belong to which subsystems. However, Bunch might produce results that conflict with this intuition for several reasons. This is addressed with a user-directed clustering feature, which enables users to cluster some modules manually, using their knowledge of the system design while taking advantage of the automatic clustering capabilities of Bunch to organize the remaining modules. Both user-directed clustering and the manual placement of omnipresent modules into subsystems have the

advantageous side-effect of reducing the search space of MDG partitions. By enabling the manual placement of modules into subsystems, these techniques decrease the number of nodes in the MDG for the purposes of the optimization and, as a result, speed up the clustering process.

Finally, once a system organization is obtained, it is desirable to preserve as much of it as possible during the evolution of the system. The integration of the orphan adoption technique into Bunch enables designers to preserve the subsystem structure when orphan modules are introduced. An orphan module is either a new module that is being integrated into the system, or a module that has undergone structural changes. Bunch moves orphan modules into existing subsystems, one at a time, and records the MQ for each of the relocations. The subsystem that produces the highest MQ is selected as the parent for the module. This process, which is linear with respect to the number of clusters in the partition, is repeated for each orphan module. Results from a case study support the added features.

The chosen additions clearly stem from real needs when modularizing software. However, two of the three operations increase the power that the user has over Bunch, thus decreasing the level of automation. Ideally the tool would be able to locate the omnipresent modules themselves, and gain the same level of expertise via a fitness function as experts, so that the user would not need to cluster anything beforehand. The last improvement, however, is truly beneficial, as hardly any software system stays intact during maintenance, and modules need to be added or modified. Automating the step of finding the optimal place for a new module is a big step towards the ideal of automating software design.

Mitchell and Mancoridis [67–69] have continued to work with the Bunch tool and have further developed the MQ metric. They define MQ as the sum of Clustering Factors for each cluster of the partitioned MDG. The Clustering Factor (CF) for a cluster is defined as a normalized ratio between the total weight of the internal edges and half of the total weight of external edges. The weight of the external edges is split in half in order to apply an equal penalty to both clusters that are connected by an external edge. If edge weights are not provided by the MDG, it is assumed that each edge has a weight of 1. The clustering factor is defined as

$$CF = \text{intra-edges}/\left(\text{intra-edges} + 1/2 * \sum(\text{inter-edges})\right).$$

The measurement is adjusted, as Mitchell and Mancoridis argue that the old MQ tended to minimize the inter-edges that exited the clusters, and not minimize the number of inter-edges in general. The representation also supports weights. This is an interesting observation, as the original definition of the MQ metric makes no distinction to whether an edge exits a cluster or not. Thus, one could ask whether the MQ metric was the sole reason for the previous results, or if other improvements besides the newly defined MQ metric also had a significant effect on obtaining the better quality results. The addition of weights is also noteworthy, as previously the problem was not considered a multi-objective one, while the addition of weights clearly indicates so.

The HC algorithm for the Bunch tool has also been enhanced. During each iteration, several options are now available for controlling the behavior of the hill-climbing algorithm. First, the neighboring process may use the first partition that it discovers with a larger MQ as the basis for the next iteration. Second, the neighboring process examines all neighboring partitions and selects the partition with the largest MQ as the basis for the next iteration. Third, the neighboring process ensures that it examines a minimum number of neighboring partitions during each iteration. For this, a threshold n is used to calculate the minimum number of neighbors that must be considered during each iteration of the process. Experience has shown that examining many neighbors during each iteration, so that $n > 75\%$, increases the time the algorithm needs to converge to a solution. This is quite intuitive, as each examination increases the run time of the algorithm, and it is not likely that simply by examining several neighbors the algorithm would suddenly find a steeper climb (i.e., converge faster).

It is observed that as n increases so does the overall runtime and the number of MQ evaluations. However, altering n does not appear to have an observable impact on the overall quality of the clustering results. A simulated annealing algorithm is also made for comparison. Although the simulated annealing implementation does not improve the MQ, it does appear to help reduce the total runtime needed to cluster each of the systems in this case study.

Mitchell and Mancoridis [69,70] continue their work by proposing an evaluation technique for clustering based on the search landscape of the graph being clustered. By gaining insight into the search landscape, the quality of a typical clustering result can be determined. The Bunch software clustering system is examined. Authors model the search landscape of each system undergoing clustering, and then analyze how Bunch produces results within this landscape in order to understand how Bunch consistently produces similar results. Studying the search landscape of any problem is very beneficial when attempting to understand why certain changes to, e.g., the fitness function or the operators, have the kind of effect they have on the results.

The search landscape is modeled using a series of views and examined from two different perspectives. The first perspective examines the structural aspects of the search landscape, and the second perspective focuses on the similarity aspects of the landscape. The structural search landscape highlights similarities and differences from a collection of clustering results by identifying trends in the structure of graph partitions. The similarity search landscape focuses on modeling the extent of similarity across all of the clustering results.

The results produced by Bunch appear to have many consistent properties. By examining views that compare the cluster counts to the MQ values, it can be noticed that Bunch tends to converge to one or two “basins of attraction” for all of the systems studied. Also, for the real software systems, these attraction areas appear to be tightly paced. An interesting observation can be made when examining the random system with a higher edge density: although these systems converged to a consistent MQ, the number of clusters varied significantly over all of the clustering runs. The percentage of intra-edges in the clustering results indicates that Bunch produces consistent solutions that have

a relatively large percentage of intra-edges. Also, the intra-edge percentage increases as the MQ values increase. It seems that selecting a random partition with a high intra-edge percentage is highly unlikely. Another observation is that Bunch generally improves the MQ of real software systems much more than that of random systems with a high edge density. The number of clusters produced compared with the number of clusters in the random starting point indicates that the random starting points appear to have a uniform distribution with respect to the number of clusters. The view shows that Bunch always converges to a "basin of attraction" regardless of the number of clusters in the random starting point.

When examining the structural views collectively, the degree of commonality between the landscapes for the systems in the case study is quite similar. Since the results converge to similar MQ values, Mitchell and Mancoridis speculate that the search space contains a large number of isomorphic configurations that produce similar MQ values. Once Bunch encounters one of these areas, its search algorithms cannot find a way to transform the current partition into a new partition with higher MQ. The main observation is that the results produced by Bunch are stable. However, the true meaning of the result is that the Bunch actually gets stuck to a local optimum, and cannot find a way to escape that local optimum. This is naturally the problem for nearly all search algorithms: a true global optimum is not even expected to be found. Doing this kind of fitness landscape study should, however, aid in designing the algorithm so that it would have a better chance of escaping the local optimum, as the fitness landscape reveals what drives the algorithm to the particular basins of attractions that it chooses.

In order to investigate the search landscape further, Mitchell and Mancoridis measure the degree of similarity of the placement of nodes into clusters across all of the clustering runs to see if there are any differences between random graphs and real software systems. Bunch creates a subsystem hierarchy, where the lower levels contain detailed clusters, and higher levels contain clusters of clusters. Results from similarity measures indicate that the results for real software systems have more in common than the results for random systems. Results with similarity measures also support the isomorphic "basin of attraction" conjecture proposed.

Mitchell et al. [71] have developed a two step process for reverse engineering the software architecture of a system directly from its source code. The first step involves clustering the modules from the source code into abstract structures called subsystems. Bunch is used to accomplish this. The second step involves reverse engineering the subsystem-level relations using a formal (and visual) architectural constraint language. Using the reverse engineered subsystem hierarchy as input, a second tool, ARIS, is used to enable software developers to specify the rules and relations that govern how modules and subsystems can relate to each other. This again gives the user the possibility to use his/her own expertise as a basis for the fitness function, so it is not based on metrics.

ARIS takes a clustered MDG as input and attempts to find the missing style relations. The goal is to induce a set of style relations that will make all of the use relations well-formed. A relation is well-formed if it does not violate any permission rule described by the style; this is called the edge repair problem. The relative quality of a proposed solution is evaluated by an objective function. The objective function that is designed into the ARIS system measures the well-formedness of a configuration in terms of the number of well-formed and ill-formed relations it contains. The quality measurement $Q(C)$ for configuration C gives a high quality score to configurations with a large number of well-formed use relations and a low quality score to configurations with a large number of ill-formed style relations or large visibility. Here, as in many other cases where some external expertise is added, the actual fitness function seems simple (only calculating sums and divisions), but much work is first needed by the user to define the input variables, here rules, for the fitness function. Again, it raises the question: what kind of automation is expected from a tool based on search algorithms? Is it good enough that the algorithm only performs a small task and expects a lot of input, or should the algorithm be better defined so that it actually diminishes the work load of the software designer instead of increasing it?

Two search algorithms have been implemented to maximize the objective function: HC and edge removal. The HC algorithm starts by generating a random configuration. Incremental improvement is achieved by evaluating the quality of neighboring configurations. A neighboring configuration C_n is one that can be obtained by a small modification to the current configuration C. The search process iterates as long as a new C_n can be found such that $Q(C_n) > Q(C)$.

The edge removal algorithm is based on the assumption that as long as there exists at least one solution to the edge repair problem for a system with respect to a style specification, the configuration that contains every possible reparable relation will be one of the solutions. Using this assumption, the edge removal algorithm starts by generating the fully reparable configuration for a given style definition and system structure graph. It then removes relations, one at a time, until no more relations can be removed without making the configuration ill-formed. A case study is performed, where the results seem promising as they give intuition to the nature of the system. This may be beneficial for novice designers, who do not have very much knowledge of the system, but it should be assumed that the developers who have to define the rules that the tool is based on already have a mature idea of the system in order to be able to define those rules.

Mahdavi et al. [72,73] show that results from a set of multiple hill climbs can be combined to locate good "building blocks" for subsequent searches. Building blocks are formed by identifying the common features in a selection of best hill climbs. This process reduces the search space, while simultaneously 'hard wiring' parts of the solution. Mahdavi et al. also investigate the relationship between the improved results and the system size.

An initial set of hill climbs is performed, and from these a set of best hill climbs is identified according to some "cut off" threshold. Using these selected best hill climbs, the common features of each solution are identified. These common features form building blocks for a subsequent hill

climb. A building block contains one or more modules fixed to be in a particular cluster, if and only if all the selected initial hill climbs agree that these modules were to be located within the same cluster. Since all the selected hill climbs agree on these choices, it is likely that good solutions will also contain these choices.

The implementation uses parallel computing techniques to simultaneously execute an initial set of hill climbs. From these climbs the authors experiment with various cut off points ranging from selecting the best 10% of hill climbs to the best 100% in steps of 10%. The building blocks are fixed and a new set of hill climbs are performed using the reduced search space. The principal research question is whether or not the identification of building blocks improves the subsequent search.

A variety of experimental subjects are used. Two types of MDGs are used: the first type contains non-weighted edges, the second type has weighted edges. The MQ values are gathered after the initial and the final climbs, and compared for difference. Statistical tests provide some evidence towards the premise that the improvement in MQ values is less likely to be a random occurrence due to the nature of the hill climb algorithm. The improvement is observed for MDGs with and without weighted edges and for all size MDGs.

Larger MDGs show more substantial improvement when the best initial fitness is compared with the best final fitness values. One reason for observing a more substantial improvement in larger MDGs may be attributed to the nature of the MQ fitness measure. To overcome the limitation that MQ is not normalized, the percentage MQ improvement of the final runs over the initial runs is measured. These statistical tests show no significant correlation between size and improvement in fitness for both weighted and non-weighted MDGs.

The increase in fitness, regardless of the number of nodes or edges, tends to be more apparent as the building blocks are created from a smaller selection of individuals. This may signify some degree of importance for the selection process.

Results indicate that the subsequent search is narrowed to focus on better solutions, that better clustering is obtained and that the results tend to improve when the selection cutoff is higher. These initial results suggest that the multiple hill climbing technique is potentially a good way of identifying building blocks. The authors also found that although there was some correlation between system size and various measures of the improvement achieved with multiple hill climbing, none of these correlations is statistically significant. These results would provide an interesting starting point to a study where the building blocks achieved with multiple hill climbs could be used to initialize the first population given to a genetic algorithm.

Harman et al. [74] experiment with fitness functions derived from measures of module granularity, cohesion and coupling for software modularization. They present a new encoding and crossover operator and report initial results based on simple component topology. The new representation allows only one representation per modularization and the new crossover operator attempts to preserve building blocks [79].

Harman et al. [74] present the problem of finding a representation for modularization so that "non-unique representations of modularizations artificially increase the search space size, inhibiting search-based approaches to the problem". In their approach, modules are numbered, and elements allocated to module numbers using a simple look-up table. Component number one is always allocated to module number one. All components in the same module as component number one are also allocated to module number one. Next, the lowest numbered component, n, not in module one, is allocated to module number two. All components into the same module as component number n are allocated to module number two. This process is repeated, choosing each lowest number unallocated component as the defining element for the module. This representation must be renormalized when components move as the result of mutation and crossover. The chosen method clearly saves resources and clarifies the search space, as there are no alternative representations for the same solution.

Harman et al.'s crossover operator attempts to preserve partial module allocations from parents to children in an attempt to promote good building blocks. Rather than selecting an arbitrary point of crossover within the two parents, a random parent is selected and one of its arbitrarily chosen modules is copied to the child. The allocated components are removed from both parents. This removal prevents duplication of components in the child when further modules are copied from one or the other parent to the child. The process of selecting a module from a parent and copying to the child is repeated and the copied components are removed from both parents until the child contains a complete allocation. This approach ensures that at least one module from the parents is preserved (in entirety) in the child and that parts of other modules will also be preserved. As it is not clarified how the modules are represented in the chromosome, it is not, however, exactly clear how risky it would be to perform traditional crossovers with the selected encoding. In fact, it seems perfectly possible to make such an encoding that supports building blocks even with the traditional operators.

The fitness function maximizes cohesion and minimizes coupling. In order to capture the additional requirement that the produced modularization has a granularity (number of modules) similar enough to the initial granularity, a polynomial punishment factor is introduced into the fitness function to reward solutions as they approach the target value for granularity of the modularization. The granularity is normalized to a percentage. The three fitness components are given equal weights.

A standard one-point crossover is also implemented for comparison. The GA with the novel crossover outperforms the one with the traditional one, although it quickly becomes trapped in local optima. This would suggest that the attempt to reserve building blocks might actually be "too strong", as the GA does not have any method to escape the local optimum. Results also show that the novel GA is more sensitive to inappropriate choices of target granularity than any other approach.

Harman et al. [75] present empirical results which compare the robustness of two fitness functions used for software module clustering: MQ is used exclusively for module clustering and EVM [80] has previously been applied

to time series and gene expression data. The clustering algorithm is based upon the Bunch algorithm [66] and redefined. Three types of MDGs were studied: real program MDGs, random MDGs and perfect MDGs.

The primary findings are that searches guided by both fitness functions degrade smoothly as noise increases, but EVM would appear to be the more robust fitness function for real systems. Searches guided by MQ behave poorly for perfect and near-perfect module dependency graphs (MDGs). The results of perfect graphs (MDGs) show however, that EVM produces clusterings which are perfect and that the clusterings produced stay very close to the perfect results as more noise is introduced. This is true both for the comparison against the perfect clustering and the initial clustering. By comparison, the MQ fitness function performs much worse with perfect MDGs. Comparing results for random and real MDGs, both fitness functions are fairly robust. Further results show that searches guided by MQ do not produce the perfect clustering for a perfect MDG but a clustering with higher MQ values. This very strongly suggests that fitness metrics indeed do not actually match what is truly desired of the solution.

These results highlight a possible weakness in MQ as a guiding fitness function for modularization searches: it may be possible to improve upon it by addressing that issue. The results show that EVM performs consistently better than MQ in the presence of noise for both perfect and real MDGs but worse for random MDGs. The results for both fitness functions are better for perfect or real graphs than random graphs, as expected. As the real programs increase in size, there appears to be a decrease in the difference between the performance of searches guided by EVM and those guided by MQ. The results show that both metrics are relatively robust in the presence of noise, with EVM being the more robust of the two.

This study is a significant indicator that fitness metrics should never be blindly trusted. The problem here is particularly curious, as the developers of the MQ metric showed the results (achieved with the aid of this metric) to actual software designers, who were reported to give positive feedback. Thus, it could be assumed that the MQ metric was based on real feedback from human designers. However, it still failed in comparison to another metric, and could not produce optimal results. These results suggest that the quality requirements for software design problems are extremely difficult to define, which in turn makes the definition of a proper fitness function a demanding task.

Antoniol et al. [76] present an approach to re-factoring libraries with the aim of reducing the memory requirements of executables. The approach is organized in two steps: the first step defines an initial solution based on clustering methods, while the second step refines the initial solution with a GA. Antoniol et al. [76] propose a GA approach that considers the initial clusters as the starting population, adopts a knowledge-based mutation function and has a multi-objective fitness function. Tests on medium and large open source software systems have effectively produced smaller, loosely coupled libraries, and reduced the memory requirement for each application.

Given a system composed by applications and libraries, the idea is to re-factor the biggest libraries, splitting them into two or more smaller clusters, so that each cluster contains symbols used by a common subset of applications (i.e., Antoniol et al. made the assumption that symbols often used together should be contained in the same library). Given that, for each library to be re-factored, a Boolean matrix *MD* is composed.

Antoniol et al. [76] have chosen to apply the Silhouette statistic [81] to compute the optimal number of clusters for each *MD* matrix. Once the number of clusters is known for each "old library", agglomerative-nesting clustering was performed on each *MD* matrix. This allows the identification of a certain number of clusters. These clusters are the new candidate libraries. When given a set of all objects contained in the candidate libraries, a dependency graph is built, and the removal of inter-library dependencies can therefore be brought back to a graph partitioning problem.

The encoding is the achieved bit-matrix, where for each matrix point $[x, y]$ has value 1 if the object y is used by the application or library defined by x, and 0 otherwise. The GA is initialized with the encoding of the set of libraries obtained in the previous step. This encoding method is well-chosen, as there is no need to make any unnecessary transformation between two encodings, and the genetic operations can be easily defined for a matrix.

The mutation operator works in two modes: normally, a random column is taken and two random rows are swapped. When cloning an object, a random position in the matrix is taken; if it is zero and the library is dependent on it, then the mutation operator clones the object into the current library. Of course the cloning of an object increases both linking and size factors, therefore it should be minimized. This GA activates the cloning only for the final part of the evolution (after 66%) of generations in their case studies. This strategy favors dependency minimization by moving objects between libraries; then, at the end, remaining dependencies are attempted to be removed by cloning objects. The crossover is a one-point crossover: given two matrices, both are cut at the same random column, and the two portions are exchanged. Population size and number of generations were chosen by an iterative procedure.

The fitness function attempts to balance three factors: the number of inter-library dependencies at a given generation, the total number of objects linked to each application that should be as small as possible, and the size of the new libraries. A unitary weight is set to the first factor, and two weights are selected using an iterative trial-and-error procedure, adjusting them each time until the factors obtained at the final step are satisfactory. The partitioning ratio is also calculated. Case study results show that the GA manages to considerably reduce the number of dependencies, while the partition ratio stays nearly the same or slightly reduced. The proposed re-factoring process allows one to obtain the smallest, loosely coupled libraries from the original biggest ones.

The selected fitness function would benefit from more enhanced techniques to deal with multi-objectivity. Also, in multi-objective problems there usually are cases when one goal may need to be emphasized at the cost of another goal. In this case there are no such tests, as the weights are simply optimized for a general case. It would be interesting to see

what kinds of results are achieved, if, e.g., the size of libraries is shown significantly more appreciation than the number of inter-library dependencies. If these cases would produce interesting modularizations, then a Pareto optimal fitness function would be good to experiment with.

Di Penta et al. [77] build on these results and present a software renovation framework (SRF), a toolkit that covers several aspects of software renovation, such as removing unused objects and code clones, and refactoring existing libraries into smaller ones. Refactoring has been implemented in the SRF using a hybrid approach based on hierarchical clustering, GAs and hill climbing, also taking into account the developer's feedback. Most of the SRF activities deal with analyzing dependencies among software artifacts, which can be represented with a dependency graph.

Software systems are represented by a system graph SG, which contains the sets of all object modules, all software system libraries, all software system applications and the set of oriented edges representing dependencies between objects. The refactoring framework consists of several steps: 1. software systems applications, libraries and dependencies among them are identified, 2. unused functions and objects are identified, removed or factored out, 3. duplicated or cloned objects are identified and possibly factored out, 4. circular dependencies among libraries are removed, or at least reduced, 5. large libraries are refactored into smaller ones and, if possible, transformed into dynamic libraries, and 6. objects which are used by multiple applications, but which are not yet organized into libraries, are grouped into new libraries. Step five, splitting existing, large libraries into smaller clusters of objects, is now studied more closely.

The refactoring of libraries is done in the SRF in the following steps: 1. determine the optimal number of clusters and an initial solution, 2. determine the new candidate libraries using a GA, 3. ask developers' feedback. The effectiveness of the refactoring process is evaluated by a quality measure of the new library organization, the Partitioning Ratio, which should be minimized.

The genome representation and mutations are as previously presented by Antoniol et al. [76]. Now, however, the developers may also give a Lock Matrix when they strongly believe that an object should belong to a certain cluster. The mutation operator does not perform any action that would bring a genome in an inconsistent state with respect to the Lock Matrix. The crossover is the one point crossover, which exchanges the content of two genome matrices around a random column.

The fitness function F should balance four factors: the number of inter-library dependencies, the total number of objects linked to each application, the size of new libraries and the feedback by developers. Thus, developer feedback is brought to the fitness function as an additional element to those already presented by Antoniol et al. [76]. The fitness function F is defined to consist of the Dependency factor DF, the Partitioning ratio PR, the Standard deviation factor SD and the Feedback factor FF. The FF is stored in a bit-matrix FM, which has the same structure of the genome matrix and which incorporates those changes to the libraries that developers suggested. Each factor of the fitness function is given a separate real, positive weight. DF is given weight 1, as it has maximum influence.

Di Penta et al. [77] report that the presented GA suffers from slow convergence. To improve its performance, it has been hybridized with HC techniques. In their experiment, applying HC only to the last generation significantly improves neither the performance nor the results, but applying HC to the best individuals of each generation makes the GA converge significantly faster. In the case study, the GA reduces dependencies of one library to about 5% of the original amount while keeping the PR almost constant. For two other libraries, a significant reduction of inter-library dependencies is obtained while slightly reducing PR in one and increasing the PR in the other. The addition of HC into GA does not improve the fitness values, since GA also converges to similar results, when it is executed on an increased number of generations and increased population size. Noticeably, performing HC on the best individuals of each generation produces a drastic reduction in convergence times. These results show that hybrid algorithms are a strong candidate when attempting to improve the results of search-based approaches.

Huynh and Cai [78] present an automated approach to check the conformance of source code modularity to the designed modularity. Design structure matrices (DSMs) are used as a uniform representation and they are automatically clustered and checked for conformance by a GA. A design DSM and source code DSM work at different levels of abstraction. A design DSM usually needs a higher level of abstraction to obtain the full picture of the system, while a source code DSM usually uses classes or other program constructs as variables labeling the rows and columns of the matrix. Given two DSMs, one at the design level and the other at the source code level, the GA takes one DSM as the optimal goal and searches for a best clustering method in the other DSM that maximizes the level of isomorphism between the two DSMs. One of the two DSMs is defined as the sample graph, and the other one as a model graph, and finally a conformance criterion is defined. This approach appears beneficial especially in the area of program comprehension and validity checking (as well as purely increasing program quality). Performing conformance checks on a large test set of programs could even produce general ideas on where the programs generally differ from the initial design.

To determine the conformance of the source code modularity to the high level design modularity the variables of the sample graph are clustered and thus a new graph is formed, which is called the conformance graph. Each vertex of the conformance graph is associated with a cluster of variables from the sample graph. The more conforming the source code modularity is to the design modularity, the closer to isomorphic the conformance graph and the model graph will be. In computing the level of isomorphism between two graphs, the graph edit distance is computed between the graphs.

With the given representation of the problem, a GA is formulated with which the goal is to find the clustering of sample graph vertices such that the conformance graph of these clustered nodes is isomorphic, or almost isomorphic, to the model graph. This is a projection. The algorithm

first creates an initial population of random projections. The fitness function is defined as $f = -D - P - \lambda - \varphi$, where D is the graph edit distance, P is a penalty, and λ and φ provide finer differentiation between mappings with the same graph edit distance. The last two functions allow the configuring of a sample graph so that it can be clustered in different ways, each corresponding to how the design targeted DSM is clustered. The dissimilarity function λ is used to calculate how separated components from each directory grouping are. If a sample graph node attribute matches a name pattern specified by the user but is not correctly mapped to the model graph vertex then the fitness of the projection is reduced through φ. Interestingly, the fitness function only measures negative aspects, quite differently to other fitness functions in modularization, which usually attempt to maximize at least some quality value.

The GA is run on two DSM models of an example software. The experiments consistently converge to produce the desired result, although the tool sometimes produces a result that is not the desired view of the source code, even though the graphs are isomorphic, i.e., the result conforms with the model. The experiment shows the feasibility of using a GA to automatically cluster DSM variables and correctly identify links between source code components and high level design components. The results support the hypothesis that it is possible to check the conformance between source code structure and design structure automatically, and this approach has the potential to be scaled for use in large software systems.

4.3. *Summarizing remarks*

The majority of the studies relating to search-based software clustering have been done with the Bunch tool, which has seen many improvements. This is very promising for other approaches to search-based design as well, as the tool has been accepted for use in the software engineering community. However, there are still many open questions in the area of software modularization. What is a proper encoding to represent a modularization problem? This question is especially highlighted by the study made by Harman et al. [74], as they point out the massive amount of redundant information in many encodings. What is a proper fitness metric for modularizations? Again, the study comparing the very popular MQ metric with another modularization metric (EVM), showed that while the metric is robust (as already validated by its developers), it can be outperformed. How can metrics be relied on then? Di Penta et al. [77] have attempted to enhance the performance of their tool by giving the developers a chance to formalize their knowledge on quality. However, defining quality as a matrix form cannot be very user-friendly.

As stated, the research on software clustering revolves quite strongly around Bunch or the MQ metric. The main exceptions to this are the studies made by Antoniol et al., [76] and Di Penta et al. [77] who use a matrix to encode the modularization and use matrix-related or metrics instead of the MQ, and Hyunh and Cai [78], who use a matrix and then turn it into a graph, and use graph related metrics to evaluate the quality of a proposed solution. Especially the approach by Hyunh and Cai [78] is significantly different to Bunch, as two modularizations are ultimately compared, while Bunch attempts to ameliorate a poor modularization without a certain goal that it is aiming towards. Thus, there is much room in search-based software clustering for alternative methods, as competition always makes each different approach strive towards even better solutions.

5. Software refactoring

5.1. *Background*

Software evolution often results in “corruption” in software design, as quality is overlooked while new features are added, or the old software should be modified in order to ensure the highest possible quality. At the same time resources are limited. Refactoring and in particular the miniaturization of libraries and applications are therefore necessary. Program transformation is useful in a number of applications including program comprehension, reverse engineering and compiler optimization. A transformation algorithm defines a sequence of transformation steps to apply to a given program and it is described as changing one program into another. It involves altering the program syntax while leaving its semantics unchanged. In object-oriented design, one of the biggest challenges when optimizing class structures using random refactorings is to ensure behavior preservation. One has to take special care of the pre- and post-conditions of the refactorings.

There are three problems with treating software refactoring as a search-based problem. First, how to determine which are the useful metrics for a given system. Second, finding how best to combine multiple metrics. Third, is that while each run of the search generates a single sequence of refactorings, the user is given no guidance as to which sequence may be best for their given system, beyond their relative fitness values.

In practice, refactoring (object-oriented software) can begin with simple restructurings of the class structure and being very close to software clustering, and then move on to a more detailed level of moving elements from one class to another. The lowest level of refactoring already deals with code, as procedures are sliced to eliminate redundancy or transformed in order to simplify the program or make it more efficient. Section 5.2 presents approaches where search-based techniques have been used to automatically achieve refactorings, as well as a study on a new method for evaluating the fitness of a refactored software. Summarizing remarks are then presented in Section 5.3, and the fundamentals of each study are collected in Table 5.

5.2. *Approaches*

Seng et al. [82] describe a methodology that computes a subsystem decomposition that can be used as a basis for maintenance tasks by optimizing metrics and heuristics of good subsystem design. GA is used for automatic decomposition. If a desired architecture is given, e.g., a layered architecture, and there are several violations, this

Table 5 – Research approaches in search-based software refactoring.

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Seng et al. [82]	Optimizing subsystem decomposition for maintenance	Model of system as a graph, extracted from source code	Genes represent subsystem candidates	Split&join, elimination and adoption	Two children from two parents, integrating crossover	Cohesion, coupling, complexity, bottlenecks and cycles	Source code extracted from resulting model	
Seng et al. [83]	Refactoring a software system with a wide set of operations	Model of system, extracted from source code, with access chains	Ordered list of refactorings	Common class structure refactorings, the list is extended with a suggested transformation	Minimize rejected, duplicated and unused methods and featureless classes and maximize abstract classes	Refactored software system	A list of suggested refactorings	
O'Keeffe and Ó Cinnéide [84]	Automating software refactoring	Software system	N/A	Restructure class hierarchy and method moves, mutations in counter-pairs in order to reverse a move	N/A	Minimize rejected, duplicated and unused methods and featureless classes and maximize abstract classes	Refactored software system	SA used as search algorithm, introducing a heuristic for weighting conflicting quality goals
O'Keeffe and Ó Cinnéide [85,86]	Automating software refactoring	System as Java source code	N/A	Refactorings regarding visibility, class hierarchy and method placement	N/A	Reusability, flexibility and understandability	Refactored code and design improvement report	Three variations of hill climbing and SA used as search algorithms
O'Keeffe and Ó Cinnéide [87,88]	Comparison between different search techniques	System as Java source code	Ordered list of refactorings (Seng et al., 2006 [83])	Common class structure refactorings, the list is extended with a suggested transformation (Seng et al. [83])	A random set of transformations from one parent chosen, the transformations of the other added to that list (Seng et al. [83]	Understandability	Refactored code and design improvement report	GA and multiple ascent hill climb implemented

Table 5 (continued)

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Qayum and Heckel [89]	Refactoring graph structure	Class diagram	N/A	A set of refactorings defined for each individual problem	N/A	Partial fitness evaluations, cost and quality	A sequence of refactorings	ACO used as search algorithm
Jiang et al. [90]	Locating dependence structures with slicing, comparing different search techniques	Source code	Two-dimensional bit matrix	A random bit flip to offspring	Multi-point crossover	Coverage and Overlap, which is divided to average and maximum	Optimal set of program slices	HC, GA, Random search and Greedy algorithm implemented; fitness function is used as a parameter
Jiang et al. [91]	Splitting procedures	Source code	Two-dimensional bit matrix	Change bit	N/A	Overlap	Optimal set of procedure slices without overlap	Greedy algorithm used
Fatiregun et al. [92]	Program refactoring on source code level	Source code	Integer vector containing transformation numbers	Standard	Standard one-point	Size of source code (LOC)	A sequence of program transformations	Random search, HC and GA are used
Williams [93]	Program parallelization	Source code	Two alternate encodings: GT includes a three symbol abbreviation of transformation and a loop number, GS includes an encoded statement	Applying one of 6 transformations, changing transformation or loop number	One-point and two-point, individual crossover points	Execution time	Transformed program	HC, GA, SA and ES implemented
Ryan and Ivan [94]	Program parallelization	Source code	Tree structure of transformations	Applying atom or loop level transformation	N/A	Execution time, correctness, loop transformation success	Transformed program, transformation sequence	GP used as algorithm
Harman and Tratt [95]	Pareto optimality used for multi-objective optimization	Software system	N/A	Move method	N/A	Coupling and standard deviation of methods per class	A sequence of refactorings	HC used as search algorithm

approach attempts to determine another decomposition that complies with the given architecture by moving classes around. Instead of working directly on the source code, it is first transformed into an abstract representation, which is suitable for a common object-oriented language.

In the GA, several potential solutions, i.e., subsystem decompositions, form a population. The initial population can be created using different initialization strategies. Before the algorithm starts, the user can customize the fitness function by selecting several metrics or heuristics as well as by changing thresholds. The model is a directed graph. The nodes of the graph can either represent subsystems or classes. Edges between subsystems or subsystems and classes denote containment relations, whereas edges between classes represent dependencies between classes. The approach is based on the Grouping GA [96], which is particularly well suited for finding groups in data. For chromosome encoding, subsystem candidates are associated with genes and the power set of classes is used as the alphabet for genes. Consequently, a gene is associated with a set of classes, i.e., an element of the power set. This representation allows a one-to-one mapping of geno- and phenotype to avoid redundant coding.

An adapted crossover operator and three kinds of mutation are used. The operators are adapted so that they are non-destructive and preserve a complete subsystem candidate as far as possible. The *split&join* mutation either divides one subsystem into two, or vice versa. The operator splits a subsystem candidate in such a way that the separation into two subsystem candidates occurs at a loosely associated point in the dependency graph. *Elimination* mutation deletes a subsystem candidate and distributes its classes to other subsystem candidates, based on association weights. *Adoption* mutation tries to find a new subsystem candidate for an orphan, i.e., a subsystem candidate containing only a single class. This operator moves the orphan to the subsystem candidate that has the highest connectivity to the orphan. The chosen mutations support reversibility, i.e., a GA can always backtrack its steps. The split&join mutation is obvious in this case, but also the adoption mutation can be seen as a reverse operation for the elimination, if a new subsystem can be created dynamically.

Initial population supports the building block theorem. Randomly selected connected components of the dependency graph are taken for half the population and highly fit ones for the rest. The crossover operator forms two children from two parents. After choosing the parents, the operator selects a sequence of subsystem candidates in both parents, and mutually integrates them as new subsystem candidates in the other parent, and vice versa, thus forming two new children consisting of both old and new subsystem candidates. Old subsystem candidates which now contain duplicated classes are deleted, and their non-duplicated classes are collected and distributed over the remaining subsystem candidates. The fitness function is defined as $f = w_1*$ cohesion $+w_2*$ coupling $+w_3*$ complexity $+ w_4*$ cycles $+ w_5*$ bottlenecks. Again the fitness function is based on the two most used metrics, cohesion and coupling, but introduces some new interesting concepts from OO design, such as cycles and bottlenecks, which are more defined than the usual general metrics.

For evaluation, a tool prototype has been implemented. Evaluation on the clustering of different software systems has revealed that results on roulette wheel selection are only slightly better than those of tournament selection. The adapted operators allow using a relatively small population size and few generations. Results from a Java case study show that the approach works well. Tests on optimizing subsets of the fitness function show that only if all criteria are optimized, are the authors able to achieve a suitable compromise with very good complexity, bottleneck and cyclomatic values and good values for coupling and cohesion. Again, as the work here is very similar to optimal software clustering, it can be questioned whether the metrics used in those studies, that mainly calculate modified values for coupling and cohesion, are actually sufficient.

Seng et al. [83] have continued their work by developing a search-based approach that suggests a list of refactorings. The approach uses an evolutionary algorithm and simulated refactorings that do not change the system's externally visible behavior. The source code is transformed into a suitable model — the phenotype. The genotype consists of the already executed refactorings. Model elements are differentiated according to the role they play in the system's design before trying to improve the structure. Not all elements can be treated equally, because the design patterns sometimes deliberately violate existing design heuristics. The approach is restricted to those elements that respect general design guidelines. Elements that deliberately do not respect them are left untouched in order to preserve the developers conscious design decisions. The notion of applying something that is known to somehow worsen the quality of a system is peculiar. In a way this is natural, as there are always trade-offs when trying to optimize conflicting quality values, but each decision should have a positive affect from some perspective. Hence, it is odd that no quality evaluator has been found that would prevent the elimination of these "deliberately violating" patterns.

The initial population is created by copying the model extracted from the source code a selected number of times. Selection for a new generation is made with tournament selection strategy. The optimization stops after a predefined number of evolution steps. The source code model is designed to accommodate several object-oriented languages. The basic model elements are classes, methods, attributes, parameters and local variables. In addition, special elements called access chains are needed. An access chain models the accesses inside a method body, because it is needed to adapt these references during the optimization. If a method is moved, the call sites need to be changed. An access chain therefore consists of a list of accesses. Access chains are hierarchical, because each method argument at a call site is modeled as a separate access chain that could possibly contain further access chains.

The model allows one to simulate most of the important refactorings for changing the class structure of a system, which are extract class, inline class, move attribute, push down attribute, pull up attribute, push down method, pull up method, extract superclass and collapse class hierarchy.

The genotype consists of an ordered list of executed model refactorings including necessary parameters. The phenotype is created by applying these model refactorings in the order that is given by the genotype to the initial source code model. Therefore the order of the model refactorings is important, since one model refactoring might create the necessary preconditions for some of the following ones.

Mutation extends the current genome by an additional model refactoring; the length of the genome is unlimited. Crossover combines two genomes by selecting the first random n model refactorings from parent one and adding the model refactorings of parent two to the genome. The refactorings from parent one are definitely safe, but not all model refactorings of parent two might be applicable. Therefore, the model refactorings are applied to the initial source code model. If a refactoring that cannot be executed is encountered due to unsatisfied preconditions, it is dropped. Seng et al. argue that the advantage of this crossover operator is that it guarantees that the externally visible behavior is not changed, while the drawback is that it takes some time to perform the crossover since the refactorings need to be simulated again. This approach is quite similar to that of Amoui et al. [32], discussed in Section 3, who approach the problem from a slightly higher level by using architectural design patterns as refactoring, but similarly search for the optimal transformation sequence.

Fitness is a weighted sum of several metric values and is designed to be maximized. The properties that should be captured are coupling, cohesion, complexity and stability. For coupling and cohesion, the metrics from Briand's [97] catalogue are used. For complexity, weighted methods per class (WMC) and number of methods (NOM) are used. The formula for stability is adapted from the reconditioning of subsystem structures. Fitness $= \sum(\text{weight}_m * (M(S)) - M_{init}(S))/M_{max}(S) - M_{init}(S)$. Before optimizing the structure the model elements are classified according to the roles they play in the systems design, e.g., whether they are a part of a design pattern.

Tests show that after approximately 2000 generations in a case study the fitness value does not significantly change anymore. The approach is able to find refactorings that improve the fitness value. Actually, this is to be expected, as it would be rather surprising if it did not improve the fitness value, as then there would be something significantly wrong with the GA. Thus, more importantly, in order to judge whether the refactorings make sense, they are manually inspected by the authors, and from their perspective, all proposed refactorings can be justified. As a second goal, the authors modify the original system by selecting 10 random methods and misplacing them. The approach successfully moves back each method at least once.

O'Keeffe and Ó Cinnéide [84] have developed a prototype software engineering tool capable of improving a design with respect to a conflicting set of goals. A set of metrics is used for evaluating the design quality. As the prioritization of different goals is determined by weights associated with each metric, a method is also described of assigning coherent weights to a set of metrics based on object-oriented design heuristics.

The presented tool, Dearthóir, is a prototype for design improvement, as it restructures a class hierarchy and moves methods within it in order to minimize method rejection, eliminate code duplication and ensure superclasses are abstract when appropriate. The refactorings are behavior-preserving transformations in Java code. The refactorings employed are limited to those that have an effect on the positioning of methods within an inheritance hierarchy. Contrary to most other approaches, this tool uses simulated annealing to find close-to-optimum solutions to this combinatorial optimization problem. In order for the SA search to move freely through the search space every change to the design must be reversible. To ensure this, pairs of refactorings have been chosen that complement each other. The refactoring pairs are: 1. move a method up or down in the class hierarchy, 2. extract (from abstract class) or collapse a subclass, 3. make a class abstract or concrete, and 4. change the superclass link of a class.

The following method is intended to filter out heuristics that cannot easily be transformed into valid metrics because they are vague, unsuitable for the programming language in use, or dependent on semantics. Firstly, for each heuristic: define the property to be maximized or minimized in the heuristic, determine whether the property can be accurately measured, and note whether the metrics should be maximized or minimized. Secondly, identify the dependencies between the metrics. Thirdly, establish precedence between dependent metrics and a threshold where necessary: prioritize heuristics. Fourthly, check that the graph of precedence between metrics is acyclic. Finally, weights should be assigned to each of the metrics according to the precedences and threshold.

The selected metrics are: 1. minimize rejected methods (RM) (number of inherited but unused methods), 2. minimize unused methods (UM), 3. minimize featureless classes (FC), 4. minimize duplicate methods (DM) (number of methods duplicated within an inheritance hierarchy), 5. maximize abstract superclasses (AS). Metrics should be appreciated so that DM $>$ RM $>$ FC $>$ AS, and UM $>$ FC. Note that the used metrics are much more specific to the needs of object-oriented design than the general structural metrics that are commonly used. Also, the heuristic of defining the weights (and the metrics) would be very beneficial for many studies, as assigning balanced weights can be a very complex task, and the dependencies between different metrics and their affect on the weights is rarely taken into account (at least so that it would be mentioned in the studies).

Most of the dependencies in the graph do not require thresholds. However, a duplicate method is avoided by pulling the method up into its superclass, which could result in the method being rejected by any number of classes. Therefore a threshold value is established for this dependency. O'Keeffe and Ó Cinnéide argue that it is more important to avoid code duplication than any amount of method rejection; therefore the threshold can be an arbitrarily high number.

A case study is conducted with a small inheritance hierarchy. The case study shows that the metric values for input and output either become better or stay the same. In the input design several classes contain clumps of methods, where as in the output design methods are spread quite evenly between the various classes. This indicates that responsibilities are being distributed more evenly among the

classes, which means that components of the design are more modular and therefore more likely to be reusable. This in turn suggests that adherence to low-level heuristics can lead to gains in terms of higher-level goals. Results indicate that a balance between metrics has been achieved, as several potentially conflicting design goals are accommodated.

O'Keeffe and Ó Cinnéide [85,86] have continued their research by constructing a tool capable of refactoring object-oriented programs to conform more closely to a given design quality model, by formulating the task as a search problem in the space of alternative designs. This tool, CODe-Imp, can be configured to operate using various subsets of its available automated refactorings, various search techniques, and various evaluation functions based on combinations of established metrics.

CODe-Imp uses a two-level representation; the actual program to be refactored is given as source code and represented as its Abstract Syntax Tree (AST), but a more abstract model called the Java Program Model (JPM) is also maintained, from which metric values are determined and refactoring preconditions are checked. The change operator is a transformation of the solution representation that corresponds to a refactoring that can be carried out on the source code.

The CODe-Imp calculates quality values according to the fitness function and effects change in the current solution by applying refactorings to the AST as required by a given search technique. Output consists of the refactored input code as well as a design improvement report including quality change and metric information.

The refactoring configuration of the tool is constant throughout the case studies and consists of the following fourteen refactorings. Push down/pull up field, push down/pull up method, extract/collapse hierarchy, increase/decrease field security, replace inheritance with delegation/replace delegation with inheritance, increase/decrease method security, made superclass abstract/concrete. During the search process alternative designs are repeatedly generated by the application of a refactoring to the existing design, evaluated for quality, and either accepted as the new current design or rejected. As the current design changes, the number of points at which each refactoring can be applied will also change. In order to see whether refactorings can be made without changing program behavior, a system of conservative precondition checking is employed.

The used search techniques include first-ascent HC (HC1), steepest-ascent HC (HC2), multiple-restart HC (MHC) and low-temperature SA. For the SA, CODe-Imp employs the standard geometric cooling schedule.

The evaluation functions are flexibility, reusability and understandability of the QMOOD hierarchical design quality model [87]. Each evaluation function in the model is based on a weighted sum of quotients on the 11 metrics forming the QMOOD (design size in class, number or hierarchies, average number of ancestors, number of polymorphic methods, class interface size, number of methods, data access metric, direct class coupling, cohesion among methods of class, measure of aggregation and measure of functional abstraction). Each metric value for the refactored design is divided by the corresponding value for the original design to give the metric change quotient. A positive weight corresponds to a metric that should be increased while a negative weight corresponds to a metric that should be decreased. It should be noted that while the complexity of the problem grew, as the program representation became more intricate, the number of refactorings (mutations) was more than doubled, this reflected on the need for a significantly more complicated fitness function. The fitness function used in the previous study only contained 5 metrics, while the current one contains 11 metrics which are grouped into 3 different fitness functions.

All techniques demonstrate strengths. HC1 consistently produces quality improvements at a relatively low cost, HC2 produces the greatest mean quality improvements in two of the six cases, MHC produces individual solutions of the highest quality in two cases and SA produced the greatest mean quality improvement in one case. Based on this it would seem that the SA is actually inferior to the different hill climbing approaches, as it only outperformed them in one measure in one test case out of the six. Combining the results of these different search algorithms would be interesting: is it possible to produce such a hybrid that would preserve the strengths from all algorithms?

Inspection of output code and analysis of solution metrics provide some evidence in favor of the use of the flexibility metric and even stronger evidence for using the understandability function. The reusability in the present form is not found suitable for maintenance because it resulted in solutions including a large number of featureless classes. As these kinds of classes are not generally accepted in OO design (apart from having "technical classes"), one might wonder whether some corrective function could be used in order to prevent featureless classes from appearing in the design. Simple pre-and post-conditions for mutations might very well help dealing with the problem. The authors conclude that both local search and simulated annealing are effective in the context of search-based software refactoring.

O'Keeffe and Ó Cinnéide [88,98] have further continued their work by implementing also a GA and a multiple ascent HC (MAHC) to the CODe-Imp refactoring tool and further testing the existing search techniques. The encoding, crossover and mutation for the GA are similar to those presented by Seng et al. [83], and the power of the tool has been increased by adding a number of different refactorings available for use in searching for a superior design.

The fitness function is an implementation of the understandability function from Bansiya and Davis's [87] QMOOD hierarchical design quality model consisting of a weighted sum of metric quotients between two designs. This choice was clearly inspired by the earlier study, where two other quality functions, flexibility and reusability, did not perform as well in terms of actual quality enhancement. This design quality evaluation function was previously found by the authors to result in tangible improvements to object-oriented program design in the context of search-based refactoring.

Results for the SA support the recommendation of low values for the cooling factor, since more computationally expensive parameters do not yield greater quality function gains.

In summary, SA has several disadvantages: it is hard to recommend a cooling schedule that will generally be effective, results vary considerably across input programs and the search is quite slow. No significant advantage in terms of quality gain was observed that would make up for these shortcomings. The GA has the advantage that it is easy to establish a set of parameters that work well in the general case, but the disadvantages are that it is costly to run and varies greatly for different input programs. Again, no significant advantage in terms of quality gain was observed that would make up for these shortcomings. Multiple-ascent HC stood out as the most efficient search technique in this study: it produced high-quality results across all the input programs, is relatively easy to recommend parameters for and runs more quickly than any of the other techniques examined. Steepest ascent HC produced surprisingly high quality solutions, suggesting that the search space is less complex than might be expected, but is slow when considering its known inability to escape local optima. Results show MAHC to outperform both SA and GA over a set of four input programs. As the genetic algorithm is the most commonly used search technique, these results should stimulate more comparisons between different algorithms. The search space for this problem was, after all, quite large, when taking into account the high number of refactorings that could be applied to a design. Thus, maybe the more refined hill climbing techniques could be compared to the GA.

Quaum and Heckel [89] apply the Ant Colony Optimization (ACO) [99] for software refactoring. The software is represented as a class diagram with methods and attributes, and the refactoring task is considered as a graph transformation problem, which makes it suitable for ACO. In order to perform ACO, five things need to be defined: 1. a set of components C and the edges between them, 2. a set of states as a sequence of components belonging to C, 3. a set of candidate solutions S, with a subset of feasible candidate solutions according to given constraints, 4. a non-empty subset (of S) of optimal solutions, and 5. an evaluation associated to each candidate solution. Based on this, Quaum and Heckel define a graph by associating the set of graph vertices to the set of proposed transformations. Edges are associated with dependencies. The pheromone and heuristic values are associated with the graph edges and are determined by partial evaluations associated with incomplete candidate solutions.

The goal is to find an optimal set of transformations. These transformations are pre-determined based on the given program (graph) and consider, e.g., moving methods and alternating the class hierarchy. An ant begins with an empty solution from the start vertex in the graph and then gradually checks the available refactoring steps in order to construct a candidate solution. Initially, any random component from C is chosen and then the partial evaluation function will guide the selection of the corresponding edge through the pheromone values. The fitness value is calculated for each feasible sequence of transformations after applying it on the source graph model, the basis for the fitness being the cost of the transformation and the quality of the result. The approach is tested on a small example system.

This approach demonstrates the use of yet another search technique, ACO, which is especially suitable for graph problems. Other choices, however, raise questions particularly on the generality of this approach. It is only tested on a small system, and all the transformations are pre-defined, and dependent on the particular system. How can this approach be generalized to be applied to any system without extensive work required to define all possible transformations of that system, which is incredibly laborious, if the system is large? Also, the details regarding fitness calculations are not very clear.

Jiang et al. [90] apply a set of search algorithms to program slicing in order to locate dependence structures. They attempt to find the subsets from all possible sets of program slices that reveal interesting dependence structures. A program is divided into slices according to program points, which are the nodes of a System Dependency Graph (SDG) [100]. In order to formulate the problem as a search problem, it is instantiated as a set cover problem. With increasing program sizes a search-based approach is extremely suitable for this type of problem.

A program is represented as a bit matrix, where rows indicate program slices and columns indicate program points. The value in point i, j, is 1 if the slice based on criterion i contains the program point j, and 0 if not. A solution should contain as many program points as possible but should have minimum overlap, i.e., slices that contain the same program points.

The fitness function is seen as a parameter to the overall approach of search-based slicing, as choosing the fitness function depends on the properties of the slice set and what the user considers as “interesting” when searching for dependencies. The fitness function is based on metrics that calculate the Coverage and Overlap of the program. Coverage measures how many program points out of all possible points the program contains. Overlap measures the number of program points within the intersection of a slicing set. It can be divided in many ways, but Jiang et al. only consider Average, which evaluates the percentage of overlapping program points based on pair-wise calculations, and Maximum, which evaluates the maximum number of overlapping points based on pair-wise calculations. Both Coverage and Overlap are given weights and then combined for the overall fitness function. Although it is said that the user can define the fitness function based on his/her own desires of what is “interesting”, it is left unclear whether the definitions must rely on the presented metrics or whether the user can build any kind of fitness function. Also, it is not clear how the properties of the slice set affect the choice of fitness function.

Jiang et al. [90] implement HC, GA, a Greedy Algorithm [101] and a Random Search algorithm. The GA uses a multi-point crossover and a standard bit change as a mutation. Elitism and rank selection are used as selection methods. For HC, a multiple restart HC is implemented in order to give it the same amount of computation time as the other algorithms. A Greedy Algorithm consists of two sets: a solution set and a candidate set, and three functions: selection, value-computing and solution function. A solution is created out of the solution set, and a candidate set represents all possible elements that might be contained in a solution. Selection chooses the most promising candidate to be added to

the solution, value-computing function gives a value for the solution and solution function checks whether the final solution has been reached. Here the initial solution set is a binary string with each bit set to 0, and the candidate solution set is made of all the slices. The value-computing function calculates the program points in a solution and the selection function chooses the one with the best coverage and smallest overlap.

An empirical study is made with six open source programs, and possible slices are collected with a separate program from each program's SDG. The program sizes vary from 37 to 1008 program points. Every other algorithm except the Greedy Algorithm was executed 100 times; the Greedy algorithm gives the same result every time and thus does not need several test runs. For the fitness function using Average Overlap, the Greedy Algorithm performs the best for all but one test case, where HC and GA perform the best. Furthermore, it is seen that for smaller programs HC outperforms GA and Random search. As the program size increases, GA starts to perform better, and wins over HC. For the second fitness function where the Maximum Overlap was used, the results are similar to with the first fitness function. However, in this case GA performs the best of the other algorithms, and HC only beats Random search on the smallest test case. The Greedy Algorithm also outperforms all others in terms of execution time. It is no surprise that the Random Search is outperformed every time. However, it is naturally a bit disappointing that the Greedy Algorithm was superior in every aspect, when compared to other search methods.

Jiang et al. [90] make another study by only using the Greedy Algorithm for six different large programs. As the previous study showed that the Greedy Algorithm outperformed all other studied search algorithms, now it is tested how efficient it is in decomposing a program into a set of slices. Results suggest that less than 20% of a program can be used to decompose the whole program or function.

Jiang et al. [91] continue by applying a Greedy Algorithm to procedure splitting. They attempt to split a procedure into two or more sub-procedures in order to improve cohesion. The Greedy Algorithm is used to find close to optimal splitting points.

A slice is represented as a bit matrix. A matrix value is depends on whether a program point (i.e., a node in the system's SDG) belongs to a certain slice. The splitting algorithm proceeds in four steps: 1. slice with respect to all nodes in SDG to find all static backward slices, 2. find sets of slices with minimum overlap, 3. recover slice statements by combining nodes that belong to a single statement, 4. make sub-procedures obtained executable.

Results indicate that more than 20% of procedures in all six programs contain independent sub-programs. Also, it would seem that most procedures are not splittable, and the ones that are, can usually be split into only 2 or 3 sub-programs. Splittability appears to correlate with the size of the program.

Fatiregun et al. [92] use meta-heuristic search algorithms to automate, or partially automate the problem of finding good program transformation sequences. With the proposed method one can dynamically generate transformation sequences for a variety of programs also using a variety of objective functions. The goal is to reduce program size, but the approach is argued to be sufficiently general that it can be used to optimize any source-code level metric. Random search (RS), hill climbing and GA are used.

An overall transformation of a program p to an improved version p' typically consists of many smaller transformation tactics. Each tactic consists of the application of a set of rules. A transformation rule is an atomic transformation capable of performing the simple alterations. To achieve an effective overall program transformation tactic, many rules may need to be applied and each would have to be applied in the correct order to achieve the desired results.

In HC, an initial sequence is generated randomly to serve as the starting point. The algorithm is restarted several times using a random sequence as the starting individual each time. The aim is to divert the algorithm from any local optimum.

Each transformation sequence is encoded as an individual that has a fixed sequence length of 20 possible transformations. An example individual is a vector of the transformation numbers. In HC, the neighbor is defined as the mutation of a single gene from the original sequence. Crossover is the standard one-point crossover. In addition to transformations, cursor moves are also used. The tournament selection is used for selecting mating parents and creating a single offspring, which replaces the worse of the parents. The authors consider optimizing the program with respect to the size of the source-code, i.e., LOC, where the aim is to minimize the number of lines of code as much as possible. This metric is quite simple, and the effects are hardly arguable, if the length of a line of code is somehow restricted.

The fitness is measured as the nominal difference in the lines of code between the source program and the new transformed program created by that particular sequence. This is evaluated by a process of five steps: 1. compute length of the input program, 2. generate the transformation sequence, 3. apply the transformation sequence, 4. compute the current length of the program, 5. compute the fitness, which is the difference between steps 1 and 4.

Results show that GA outperforms both RS and HC. In cases where RS outperformed GA and HC, it was noticed that GA and HC are not "moving" towards areas where potential optimizations could be. Analyzing the GA, the authors believe that the GA potentially kills off good subsequences of transformations during crossover. These results are interesting as this would indicate that the selected (standard) crossover would not support the preservation of building blocks. As discussed in Section 4, it may be that also the encoding could be improved to preserve building blocks. All in all, examining the fitness landscape and rethinking the encoding and crossover operators may be able to improve the results achieved with the GA.

Williams [93] implements several search algorithms in his REVOLVER system that make program transformations in order to parallelize the program and thus lessen the execution time. The idea is to transform loops in different ways, and as loops are the core of the approach, they are numbered. HC, SA and GA are used, and most interestingly, two different encodings are experimented with.

In the first encoding, Gene-Transformation (GT), each gene represents a transformation that is applied to the system. The gene contains information as to what transformation

is applied, and the number of the loop it is applied to. Three different mutations can be used: changing the transformation, changing the loop number or changing both (i.e., the entire gene). Both one-point and two-point crossovers are implemented. However, in the one-point crossover, the crossover points for the parent chromosomes are chosen individually for each parent, as they might be of unequal length. This approach is applied to HC, SA and GA.

In the second encoding, Gene-Statement (GS), each gene represents a statement in the program, e.g., an if- or a do-statement, and the chromosome thus represents the program as a sequence of statements. The mutations that are applied are the chosen operations on loops, and applying them to the program. This is actually quite odd, as only loop related transformations are used, but there are only loops in some of the genes. Note, that a mutation will alter the program, as, say, combining two loops will remove the statement representing one of them, and thus shortens the chromosome by one gene. No crossover is used in this representation, and the used algorithms are HC and evolutionary strategy (ES), which is basically a GA, i.e., it has a population and selection, but without the crossover.

The fitness function for both approaches is the actual execution time of the transformed program, and tournament selection is used. In the tests the population size was only 5 for the algorithms with populations, and the number of generations only 50. These parameters seem incredibly low, as there is very little room for versatility in the population, and there is very little time for development also. Thus, one wonders whether the benefits of the GA are truly used in this approach.

Test results on five programs show that the ES and HC with the GS encoding outperformed all other algorithms. The traditional GA appeared the worst. These results further suggest that the population parameter chosen for the traditional GA should be revised, as the GA cannot use its full potential. Interestingly, ES, which also had a population, performed the best. The strength of the GS encoding is also very interesting, considering there is much information in the genes that cannot be mutated. However, ES did not have a crossover, and thus choosing parents is not an issue for this algorithm. All in all, the algorithms were able to improve the execution times significantly.

Ryan and Ivan [94] have taken a rather different approach to program parallelization, as they encode the program in tree form and use genetic programming as the search algorithm. They use GP in an unusual way, as it does not actually "program", but searches for the optimal transformations for the program, thus making this study a design problem.

The program is considered as a sequence of instructions. The actual tree given by the GP then comes from examining the atoms representing the instructions, and deciding on transformations based on the type of the instruction. The GP works in two modes: atom mode and loop mode. Each step begins in atom mode, and if the found instruction is a loop, the mode is switched. In atom mode, there are three classes of transformations. The transformations in the first class split the sequence of instructions according to a given percent, thus forking the execution of a program. The ones in the second class also split the sequence of instructions, but with less effect, as the split point is always either after the first of before the last instruction. The last class delays the execution of the program. Each atom mode transformation is an internal node in a tree, and takes as input the program segment before passing it onto the next transformation. The program segment ultimately diminishes to one atom as transformations are applied. In loop mode the idea is to parallelize each loop by executing each iteration on a different processor, unfortunately, though, this raises issues with data dependencies. A significant operator in loop mode is loop fusion, which combines consecutive loops.

The fitness function is a combination of fitness calculations from the atom mode and the loop mode. For the atom mode the fitness is the execution time and the correctness of the program. For loop mode the fitness is the number of successes for applied loop operators. The initial results are promising; the approach is able to parallelize programs and thus ameliorate them in terms of execution time.

The approach of Ryan and Ivan [94] appears quite similar to that of Williams [93] in terms of the choosing loops as a key ingredient in the mutations. However, Ryan and Ivan have taken atom transformations into account as traditional mutations, while Williams has chosen to deal with non-loop structures only at the encoding stage. The fitness function for both approaches is basically the same, as execution time is the most important factor. It would be interesting to study the problem of program parallelization also in terms of other quality factors and as a larger problem in the context of, e.g., distributed systems.

Harman and Tratt [95] show how Pareto optimality can improve search based refactoring, making the combination of metrics easier and aiding the presentation of multiple sequences of optimal refactorings to users. Intuitively, each value on a Pareto front maximizes the multiple metrics used to determine the refactorings. Through results obtained from three case studies on large real-world systems, it is shown how Pareto optimality allows users to pick from different optimal sequences of refactorings, according to their preferences. Moreover, Pareto optimality applies equally to sub-sequences of refactorings, allowing users to pick refactoring sequences based on the resources available to implement those refactorings. Pareto optimality can also be used to compare different fitness functions, and to combine results from different fitness functions.

Harman and Tratt [95] use the move method refactoring presented by Seng et al. [83]. Three systems are used in the case study, all non-trivial real-world systems. The search algorithm itself is a non-deterministic non-exhaustive hill climbing approach. A random move method refactoring is chosen and applied to the system. The fitness value of the updated system is then calculated. If the new fitness value is worse than the previous value, the refactoring is discarded and another one is tried. If the new fitness value is better than the previous, the refactoring is added to the current sequence of refactorings, and applied to the current system to form the base for the next iteration. A cut-off point is set for checking neighbors before concluding that a local maximum is reached. The end result of the search is a sequence of refactorings and a list of the before and after values of the various metrics involved in the search.

Two metrics are used to measure the quality: coupling and standard deviation of methods per class (SDMPC). Coupling (CBO) is from Briand's [97] catalogue. The second metric, SDMPC, is used to act as a 'counter metric' for coupling. An arbitrary combination of the metrics is used, the fitness function being SDMPC $*$ CBO. The new fitness function improves the CBO value of the refactored system while also improving the SDMPC of the system. All the points on a Pareto front are, in isolation, considered equivalently good. In such cases, it might be that the user may prefer some of the Pareto optimal points over others.

The concept of a Pareto front is argued to make as much sense with subsets of data as it does for complete sets. Harman and Tratt [95] also stress the importance of knowing how many runs a search-based refactoring system will need to achieve a reasonable Pareto front approximation. Furthermore, developers are free to execute extra runs of the system if they feel they have not yet achieved points of sufficient quality on the front approximation. Pareto optimality allows determining whether one fitness function is subsumed by another: broadly speaking, if fitness function f produces data which, when merged with the data produced from function f', contributes no points to the Pareto front then we know that f is subsumed by f'. Although it may not be immediately apparent, Pareto optimally confers a benefit potentially more useful than simply determining whether one fitness function is subsumed by another. If two fitness functions generate different Pareto optimal points, then they can naturally be combined to a single front. Pareto optimality is shown to have many benefits for search-based refactoring, as it lessens the need for "perfect" fitness functions. This would make Pareto optimality an approach that should be considered for any optimization problem with conflicting goals.

5.3. *Summarizing remarks*

The approaches to search-based refactorings can be divided into the following groups: refactoring the program at class level, refactoring the program at procedure level, and refactoring pieces of code. The most studies have been performed on refactoring at class level, and they are all quite similar, and actually end up using the same operations for the search algorithm. For the other aspects only one or two studies have been made, and this suggests that there is much room for competing approaches. The most advanced results have been achieved with refactorings at class level, while studies in program transformations have achieved both good and not so good results.

When examining the refactoring problems, one notable characteristic is that Seng et al. [83] attempt to preserve building blocks from the very beginning, and several other studies have later built on the operators introduced by them. The mutation selection by Seng et al. [83] also appears popular. The complexity of the refactoring problem at class level was most pointedly demonstrated by O'Keeffe and Ó Cinnéide [85,86], who had a list of 14 mutations and 11 metrics, and Quaum and Heckle [89], who had to pre-define mutations according to the specific system. Considering that there can be even more general refactorings in addition to those presented by O'Keeffe and Ó Cinnéide, and that they could be combined with system specific mutations, the search space for an optimal refactoring sequence will soon become incredibly large.

The approaches to search based refactoring also seem advanced in the aspect that there have already been several studies that compare different search algorithms and fitness functions. As for the search algorithms, different hill climbing applications are clearly very efficient and able to produce high quality results. Interestingly, simulated annealing has been outperformed by other algorithms, although one might argue that it is more "sophisticated" than at least the basic hill climbing. All in all, there are very few approaches that use simulated annealing, and no breaking results have been achieved with it. The studies in fitness functions further support the notion of complexity in this problem area. O'Keeffe and Ó Cinnéide [84] have considered the problem of finding an appropriate fitness function so important that they have developed a heuristic for balancing different weights, and Harman et al. [95] have introduced the Pareto optimality concept to this field, as software design is indeed an area where trade-offs and compromises need to be made. As for the other studies, the variety of metric quality evaluators shows that a refined method for deciding on an appropriate fitness function is truly needed. The only area where consensus can be found is program transformations, where quality can quite simply be measured in terms of run time and correctness or size of the program.

6. Software quality

6.1. *Background*

Software quality assessment has become an increasingly important field. The complexity caused by object-oriented methods makes the task more important and more difficult. An ideal quality predictive model can be seen as the mixture of two types of knowledge: common knowledge of the domain and context specific knowledge. In existing models, one of the two types is often missing. During its operating time, a software system undergoes various changes triggered by error detection, evolution in the requirements or environment changes. As a result, the behavior of the software gradually deteriorates as modifications increase. This quality slump may go as far as the entire software becoming unpredictable.

Software quality is a special concern when automatically designing software systems, as the quality needs to be measured with metrics and in pure numerical values. The use of metrics may even be argued, as they cannot possibly contain all the knowledge that an experienced human designer has. Sahraoui et al. [102] have investigated whether some object-oriented metrics can be used as an indicator for automatically detecting situations where a particular transformation can be applied to improve the quality of a system. The detection process is based on analyzing the impact of various transformations on these object-oriented metrics using quality estimation models.

Sahraoui et al. [102] have constructed a tool which, based on estimations on a given design, suggests particular

transformations that can be automatically applied in order to improve the quality as estimated by the metrics. Roughly speaking, building a quality estimation model consists of establishing a relation of cause and effect between two types of software characteristics. Firstly, internal attributes which are directly measurable, such as size, inheritance and coupling, and secondly, quality characteristics which are measurable after a certain time of use such as maintainability, reliability and reusability. To study the impact of the global transformations on the metrics, first the impact of each elementary transformation is studied and then the global impact is derived. A case study is used for the particular case of the diagnosis of bad maintainability by using the values of metrics for coupling and inheritance as symptoms. Based on the results of this study, Sahraoui et al. [102] argue that using metrics is a step toward the automation of quality improvement, but that experiments also show that a prescription cannot be executed without the validation of a designer/programmer.

The use of evolution metrics for fitness functions has especially been studied [103,104]. If one looks at the whole process of detecting flaws and correcting them, metrics can help automating a large part of it. However, the results of the experiments show that a prescription cannot be executed without the validation of a designer or programmer. This approach cannot capture all the context of an application to allow full automation.

Some approaches regarding software quality have also been made with search-based techniques and are detailed in Section 6.2. Bouktif et al. [105,106] aim at predicting software quality of object-oriented systems with GAs, and Vivanco and Jin [108] have implemented a GA to identify possible problematic software components. Bouktif et al. [107] have also implemented a SA to combine different quality prediction models. Summarizing remarks are presented in Section 6.3, and the fundamentals of each approach are collected in Table 6.

6.2. *Approaches*

Bouktif et al. [105,106] study the prediction of stability at object-oriented class level and propose two GA based approaches to solve the problem of quality predictive models: the first approach combines two rule sets and the second one adapts an existing rule set. The predictive model will take the form of a function that receives as input a set of structural metrics and an estimation of stress, and produces as output a binary estimation of the stability. Here, stress represents the estimated percentage of added methods in a class between two consecutive versions.

The model encoding for the GA that combines rule sets is based on a decision tree. The decision tree is a complete binary tree where each inner node represents a yes-or-no question, each edge is labeled by one of the answers, and terminal nodes contain one of the classification labels from a predetermined set. The decision making process starts at the root of the tree. When the questions at the inner nodes are of form "*Is* $x > a$?", the decision regions of the tree can be represented as a set of isothetic boxes in an n-dimensional space (n = number of metrics). For the GA representation, these boxes are enumerated in a vector. Each gene is a (box, label) pair, and a vector of these pairs is the chromosome. The complexity of quality as a concept is directly shown in the complexity of the encoding. No simple integer vector can be used to represent quality estimations. An interesting research question is to determine what is the minimal information needed in order to evaluate or predict quality.

Mutation is a random change in the genes that happens with a small probability. In this problem, the mutation operator randomly changes the label of a box. To obtain an offspring, a random subset of boxes from one parent is selected and added to the set of boxes of the second parent. The size of the random subset is υ times the number of boxes of the parent, where υ is a parameter of the algorithm. By keeping all the boxes of one of the parents, completeness of the offspring is automatically ensured. To guarantee consistency, the added boxes are made predominant (the added boxes are "laid over" the original boxes). A level of predominance is added as an extra element to the genes. Each gene is now a three-tuple (box, label, level). The boxes of the initial population have level 1. Each time a predominant box is added to a chromosome, its level is set to 1 plus the maximum level in the hosting chromosome. To find the label of an input vector x (a software element), first all the boxes containing x are found, and x is assigned the label of the box that have the highest level of predominance.

To measure the fitness a correctness function is used; the function calculates the number of cases that the rule correctly classifies divided by the total number of cases that the rule classifies. The correctness function is defined as $C = 1 -$ training error. By using the training error for measuring the fitness, it is found that the GA tended to "neglect" unstable classes. To give more weight to data points with minority labels, Youden's [109] J-index is used. Intuitively, the J-index is the average correctness per label. If one has the same number of points for each label, then $J = C$. As seen, the actual fitness evaluations for quality seem simple, which is surprising when compared to the complicated metric combinations used to evaluate quality in all the various GA implementations already presented. However, here the most work is needed for defining the rules that need to be satisfied and questions that need to be answered.

With a GA for adapting a rule set, an existing rule set is used as the initial population of chromosomes, each rule of the rule set being a chromosome and each condition in the rule as well as the classification label being a gene. Each chromosome is attributed a fitness value, which is $C * t$, where t is the fraction of cases that the rule classifies in the training set. The weight t allows rules to be given that cover a large set of training cases a higher chance of being selected.

Parents for crossover are selected with the roulette wheel method. A random cut point is generated for each parent, i.e., the cut-points are different for each parent. Otherwise, the operation is a traditional one-point crossover. By allowing chromosomes within a pair to be cut at different places, a wider variety is allowed with respect to the length of the chromosomes. The chromosomes are then mutated. The mutation of a gene consists of changing the value to which the attribute encoded in the gene is compared to a value chosen randomly from a predefined set of values for the

Table 6 – Studies in search-based software quality enhancement.

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Bouktif et al. [105,106]	Combining two rule sets vs. adapting a rule set with GA in quality prediction models	Decision tree	Combination: box, label -pairs from decision tree Adaptation: one rule is one chromosome, each condition in the rule is a gene	Combination: change of label Adaptation: change value of attribute encoding	Combination: a random set of boxes from one parent added to the other and level of predominance added to gene (box, label, level) Adaptation: standard one-point, parents selected with roulette-wheel method	Correctness	Optimal rule set	
Bouktif et al. [107]	Combining software quality prediction models, i.e., experts	Set of example models and context data	Range and conditional probabilities	Modify range or probability or add or remove an expert	N/A	Correctness	Optimal model combined of sub-optimal models	SA used
Vivanco and Jin [108]	Identification of complex components	Software system	N/A	N/A	N/A	OO metrics	Classes divided according to complexity levels	

attribute (or class label, in case the last gene is mutated). The new chromosomes are scanned and trimmed to get rid of redundancy in the conditions that form the rules that they encode. Inconsistent rules are attributed a fitness value of 0 and will eventually die. A fixed population size is maintained. Elitism is performed when the population size is odd. This consists of copying one or more of the best chromosomes from one generation to the next. Before passing from one generation to another, the performance of combined rules to one rule set is evaluated.

In the experimental setting, to build experts (that simulate existing models), stress and 18 metrics (belonging to coupling, cohesion, complexity and inheritance) are used. Eleven object-oriented systems are used to “create” 40 experts. For the combining GA, the elitist strategy is used, where the entire population apart from a small number of fittest chromosomes is replaced. The test results show that the approach of combining experts can yield significantly better results than using individual models. The adaptation approach does not perform as well as the combination, although it gave a slight improvement over the initial model in one case. The authors believe that using more numerous and real experts on cleaner and less ambiguous data, the improvement will be more significant. It is quite inspiring that the approach of combining experts produced the more promising results. If it can be assumed that experts in both initial populations have the same amount of knowledge, it would seem that merely adapting an expert would be a smaller task to perform than successfully combining the knowledge from two different experts. Thus the results are very positive when considering what the GA is capable of.

Bouktif et al. [107] have continued their research by applying simulated annealing to combine experts. Their approach attempts to reuse and adapt quality predictive models, each of which is viewed as a set of expertise parts. The search then aims to find the best subset of expertise parts, which forms a model with an optimal predictive accuracy. The SA algorithm and a GA made for comparison were defined for Bayesian classifiers (BCs), i.e., probabilistic predictive models.

An optimal model is built of a set of experts, each of which is given a weight. Each individual, i.e., chunk, of expertise is presented by a tuple consisting of an interval and a set of conditional probabilities. Transitions in the neighborhood are made by changing probabilities or interval boundaries. A transition may also be made by adding or deleting a chunk of expertise. The fitness function is the correctness function.

For evaluation, the SA needs two elements as inputs: a set of existing experts and a representative sample of context data. Results show a considerable improvement in the predictive accuracy, and the results produced by the SA are stable. The values for GA and SA are so similar that the authors do not see a need to value one approach over the other. Results also show that the accuracy of the best produced expert increases as the number of reused models increases, and that good chunks of expertise can be hidden in inaccurate models. Again, the results achieved with SA encourage further usage of different search algorithms apart from GA, or even combining and making more hybrid approaches in order to increase quality in search based approaches to software design.

Vivanco and Jin [108] present initial results of using a parallel GA as a feature selection method to enhance a predictive model’s ability to identify cognitively complex components in a Java application. Linear discriminant analysis (LDA) can be used as a multivariate predictive model.

It is theorized that the structural properties of modules have an impact on the cognitive complexity of the system, and further on, that modules that exhibit high cognitive complexity result in poor quality components. Again, this is in line with the assumption already made by Lutz [42], that the simpler a design, the better. A preliminary study is carried out with a biomedical application developed in Java. Experienced program developers are asked to evaluate the system. Classes labeled as low are considered easy to understand and use, while a high ranking implied the class is difficult to fully comprehend and would likely take considerably much more effort to maintain. Source code measurements, 63 metrics for each Java class, are computed using a commercial source code inspection application. To establish a baseline, all the available metrics are used with the predictive model. The Chidamber and Kemerer [45] metrics suite is used to determine if the model would improve. Finally, the GA is used to find alternate metrics subsets. Using the available metrics with LDA, less than half of the Java classes are properly classified as difficult to understand. The CK metrics suite performs slightly better. Using GA, the LDA predictive model has the highest performance using a subset of 32 metrics. The GA metrics correctly classify close to 100% of the low, nearly half of the medium and two thirds of the high complexity classes.

Vivanco and Jin [108] are most interested in finding the potentially problematic classes with high cognitive complexity. A two-stage approach is evaluated. First, the low complexity classes are classified against the medium/high complexity classes. The GA driven LDA highly accurately identifies the low and medium/high complexity classes with a subset of 24 metrics. When only the medium complexity classes are compared to high complexity, a GA subset of 28 metrics results in extremely high accuracy for the medium complexity classes and in identifying the problematic classes. In all GA subsets, metrics that cover Halstead complexity, coupling, cohesion, and size are used, as well as program readability metrics such as comment to code ratios and the average length of method names.

This study is extremely interesting as it ties known software metrics to human expertise and compares how metrics perform when trying to correctly classify objects. It is noteworthy that from 63 different metrics the optimal outcome was achieved with 24–32 metrics, which is less than half of all metrics available. Although there is naturally overlap between different metrics, it is interesting to see that many of them do not seem to correctly evaluate the program. The found metrics cohesion, coupling and complexity support the current fitness function choices to a certain points. However, many fitness functions only calculate 2–5 different metrics, while the optimum was reached with over 20. In addition, several metrics need the source code, and thus make them unsuitable for more high-level problems.

6.3. Summarizing remarks

The presented studies on software quality estimation show that correctly evaluating software is anything but easy. However, although the number of studies is small, they are all very recent, and thus shows promise that search-based approaches can also be used in this sub-area of software design. Finding a search algorithm for quality estimation can also be seen as a developed way of tackling the problem of finding an optimal fitness function. In other words, in the future it might be possible to use a fitness function (i.e., a search algorithms) to find an optimal fitness function for each individual software design problem. Using search algorithms for quality estimations, the current fitness function, is the first step in this direction.

7. Future work

From the search-based approaches presented here, software clustering and software refactoring (i.e., re-design) appear to be at the most advanced stage. Thus, most work is needed with actual architecture design, starting from requirements and not a ready-made system. Also, search-based application of, e.g., design patterns, should be investigated more. Another branch of research should be focused on quality metrics. So far the quality of a software design has mostly been measured with cohesion and coupling, which mostly conform to the quality factors of efficiency and modifiability. However, there are many more quality factors, and if an overall stable software system is desired, more factors should be taken into account in evaluation, such as reliability and stability. Also, as demonstrated with the MQ metric in Section 4, metrics that have seemed good in the beginning may prove to be inadequate when investigated further. Fortunately, it seems that most of the work presented here is the result of developing research that is still continuing. The following research questions should and could very well be answered in the foreseeable future:

– What kind of architectural decisions are feasible to do with search-based techniques?

Research with search-based software architecture design is at an early stage, and not all possible architecture styles and design patterns have been tested. Some architectural decisions are more challenging to implement automatically than others, and in some cases it may not be possible at all. The possibilities should be mapped to effectively research the extent of search-based designs capabilities.

– What is a sufficient starting point for software architecture design with search-based techniques?

So far requirements with a limited set of parameters have been used to build software architecture, or a ready system has been improved. Some design choices need very detailed information regarding the system in order to effectively evaluate the change in quality after implementing a certain design pattern or architecture style. The question of what information is needed for correct quality evaluation is by no means easily answered.

– What would be optimal representation, crossover and mutation operators regarding the software modularization problem?

Much work has been done with software modularization, and the chromosome encoding, crossover and mutation operators vary greatly. Optimal solutions would be interesting to find. As discussed throughout the survey, the chosen encoding significantly affects the result of mutation and crossover operations and also has a big impact on run time for the algorithm. There are also several options for crossover, where some maintain building blocks better than others.

– What would be optimal representation, crossover and mutation operators regarding the software refactoring problem?

Much research has been done with software refactoring, and the chromosome encoding, crossover and mutation operators vary greatly. Especially the set of mutations is interesting, as they define how greatly the software can be refactored. An optimal encoding might enable a larger set of mutations, thus giving the search-based algorithm a larger space to search for optimal solutions.

– What metrics could be seen as a "standard" for evaluating software quality?

The evaluation of quality, i.e., the fitness function, is a crucial part of evolutionary approaches to software engineering. Some metrics, e.g., coupling and cohesion, have been widely used to measure quality improvements at different levels of design. However, these metrics only evaluate a small portion of quality factors, and there are several versions of even some very "standard" metrics. Metrics by, e.g., Briand [83] and Chidamber and Kemerer [45] can be considered as some kind of standards. However, all software metrics are constantly subjected to criticism, as their correctness is challenged. Thus, in the author's view, as there are several versions of even the most common metrics and there is no agreement that metrics even measure the right things at the moment, no metric set can currently be seen as standard. Thus, a well-validated metric set would be extremely beneficial, if it is possible to conduct such a set. It very well may be that the present metrics simply don't suffice, and in that case other directions must be taken to evaluate quality, as has already been demonstrated in some of the work covered in this survey.

– How can metrics be grouped to achieve more comprehendible quality measures?

Metrics achieve clear values, but if a human designer attempts to use a tool in the design process, notions such as "efficiency" and "modifiability" are more comprehendible than "coupling" and "cohesion". Thus, being able to group sets of metrics to correspond to certain real-world quality values would be beneficial when making design tools available for common use.

8. Conclusions

This survey has presented on-going research in the sub-fields of search-based software design. There has been much progress in the sub-fields of software modularization and refactoring, and very promising results have been achieved. A more complex problem is automatically designing software architecture from requirements, but some initial steps have already been taken in this direction as well. Fig. 2 shows

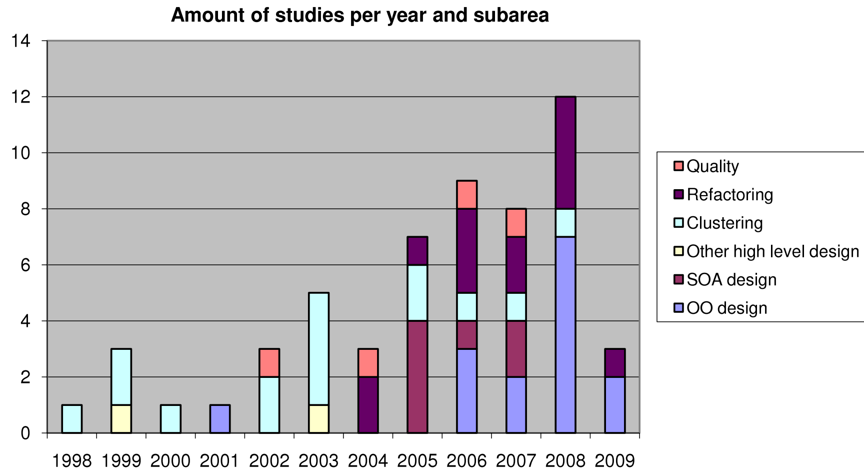


Fig. 2 – Timeline for studies in search-based design.

the timeline of the presented studies, and it very effectively demonstrates the increasing interest in the area during very recent years. There has been immense increase in the area of OO design and refactoring, while clustering, the first application in the area, has not sparked new research interest.

The surveyed research shows that metrics, such as cohesion and coupling can accurately evaluate some quality factors, as the achieved, automatically improved designs, have been accepted by human designers. However, many authors also report problems: the quality of results is not as high as wished or expected, and many times the blame is placed on less than optimal encoding and crossover operators. Extensive testing of different encoding options is practically infeasible, and thus inspiration could be found in those solutions that have produced the most promising results. As a whole, software (re-)design seems to be an appropriate field for the application of meta-heuristic search algorithms, and there is much room for further research.

Acknowledgements

The author would like to thank Professor Erkki Mäkinen for his helpful comments when writing this survey. This work was partially done for the Darwin project, funded by the Academy of Finland.

REFERENCES

[1] SSBSE, 2010, http://www.ssbse.org, checked 17.2.2010.
[2] M. Harman, The current state and future of search based software engineering, in: Proceedings of the 2007 Future of Software Engineering, FOSE'07, 2007, pp. 342–357.
[3] J. Clarke, J.J. Dolado, M. Harman, R.M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, M. Shepperd, Reformulating software engineering as a search problem, IEE Proceedings - Software 150 (3) (2003) 161–175.
[4] M. Harman, B.F. Jones, Search based software engineering, Information and Software Technology 43 (14) (2001) 833–839.
[5] P. McMinn, Search-based software test data generation: a survey. Software Testing, Verification and Reliability 14 (2) (2004) 105–156.
[6] T. Mantere, J.T. Alander, Evolutionary software engineering: a review, Applied Soft Computing 5 (3) (2005) 315–331.
[7] W. Afzal, R. Torkar, R. Feldt, A systematic mapping study on non-functional search-based software testing, in: Proceedings of SEKE 2008, 2008, pp. 488–493.
[8] W. Afzal, R. Torkar, R. Feldt, A systematic review of search-based testing for non-functional system properties, Information and Software Technology 51 (6) (2009) 57–83.
[9] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper, Testability transformation, IEEE Transactions on Software Engineering 30 (1) (2004) 3–16.
[10] S.S. Yau, J.-P. Tsai, A survey of software design techniques, IEEE Transactions on Software Engineering 12 (6) (1986) 713–721.
[11] R.J. Wirfs-Brock, R.E. Johnson, Surveying current research in object-oriented design, Communications of the ACM 33 (9) (1990) 104–124.
[12] D. Budgen, Software Design, Pearson, 2003.
[13] M. Harman, S.A. Ansouri, J. Zhang, Search based software engineering: a comprehensive review, Technical report TR-09-03, King's College, London, United Kingdom, 2009.
[14] M. Harman, J. Wegener, Getting results with search-based approaches to software engineering, in: Proceedings of the ICSE'04, 2004, pp. 728–729.
[15] J.H. Holland, Adaption in Natural and Artificial Systems, MIT Press, Ann Arbor, 1975.
[16] J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.
[17] M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, 1996.
[18] S.N. Sivanandam, S.N. Deepa, Introduction to Genetic Algorithms, Springer, 2007.
[19] C.R. Reeves (Ed.), Modern Heuristic Techniques for Combinatorial Problems, McGraw-Hill, 1995.
[20] F.W. Glover, G.A. Kochenberger (Eds.), Handbook of Metaheuristics, International Series in Operations Research & Management Science, vol. 57, Springer, 2003.

[21] K. Deb, Evolutionary algorithms for multicriterion optimization in engineering design, in: Proc. Evolutionary Algorithms in Engineering and Computer Science, EUROGEN'99, pp. 135–161.

[22] C. Fonseca, P. Fleming, An overview of evolutionary algorithms in multi-objective optimization, Evolutionary Computation 3 (1) (1995) 1–16.

[23] M. Zlochin, M. Birattari, N. Meuleau, M. Dorigo, Model-based search for combinatorial optimization: a critical survey, Annals of Operations Research 131 (2004) 373–395.

[24] M. Shaw, D. Garlan, Software Architecture — Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[25] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[26] F. Losavio, L. Chirinos, A. Matteo, N. Lévy, A. Ramdane-Cherif, ISO quality standards for measuring architectures, The Journal of Systems and Software 72 (2004) 209–223.

[27] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, 1998.

[28] M. Bowman, L.C. Briand, Y. Labiche, Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms, Technical report SCE-07-02, Carleton University, 2008.

[29] C.L. Simons, I.C. Parmee, (a) Single and multi-objective genetic operators in object-oriented conceptual software design, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'07, 2007, pp. 1957–1958.

[30] C.L. Simons, I.C. Parmee, (b) A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design, Engineering Optimization 39 (5) (2007) 631–648.

[31] C.L. Simons, I.C. Parmee, User-centered, evolutionary search in conceptual software design, in: Proceedings of the IEEE Congress on Evolutionary Computarion, CEC'08, 2008, pp. 869–876.

[32] M. Amoui, S. Mirarab, S. Ansari, C. Lucas, A genetic algorithm approach to design evolution using design pattern transformation, International Journal of Information Technology and Intelligent Computing 1 (1–2) (2006) 235–245.

[33] O. Räihä, K. Koskimies, E. Mäkinen, (a) Genetic synthesis of software architecture, in: Proceedings of the 7th International Conference on Simulated Evolution and Learning, SEAL'08, Australia, in: LNCS, vol. 5361, 2008, pp. 565–574.

[34] O. Räihä, K. Koskimies, E. Mäkinen, T. Systä, (b) Pattern-based genetic model refinements in MDA, Nordic Journal of Computing 14 (4) (2008) 338–355.

[35] O. Räihä, K. Koskimies, E. Mäkinen, Scenario-based genetic synthesis of software architecture, in: Proceedings of the 4th International Conference on Software Engineering Advances, ICSEA'09, Portugal, IEEE Computer Society Press, 2009, pp. 437–445.

[36] M. Kessentini, H. Sahraoui, M. Boukadoum, Model transformation as an optimization problem, in: Proceedings of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems, MODELS'08, 2008, pp. 159–173.

[37] D. Kim, S. Park, Dynamic architectural selection: a genetic algorithm approach, in: Proceedings of the 1st Symposium on Search-Based Software Engineering, 2009, pp. 59–68.

[38] T. Bodhuin, M. Di Penta, L. Troiano, A search-based approach for dynamically re-packaging downloadable applications, in: Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research, CASCON'07, 2007, pp. 27–41.

[39] N. Gold, M. Harman, Z. Li, K. Mahdavi, A search based approach to overlapping concept boundaries, in: Proceedings of the 22nd International Conference on Software Maintenance, ICSM 06, USA, 2006, pp. 310–319.

[40] H. Goldsby, B.H.C. Chang, Avida-mde: a digital evolution approach to generating models of adaptive software behavior, in: Proceedings of the Genetic Evolutionary Computation Conference, GECCO'08, 2008, pp. 1751–1758.

[41] H. Goldsby, B.H.C. Chang, P.K McKinley, D. Knoester, C.A. Ofria, Digital evolution of behavioral models for autonomic systems, in: Proceedings of 2008 International Conference on Autonomic Computing, 2008, pp. 87–96.

[42] R. Lutz, Evolving good hierarchical decompositions of complex systems, Journal of Systems Architecture 47 (2001) 613–634.

[43] L.J. Fogel, A.J. Owens, M.J. Walsh, Artificial Intelligence through Simulated Evolution, Wiley, 1966.

[44] R.C. Martin, Design Principles and Design Patterns, 2000, available at: http://www.objectmetor.com.

[45] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–492.

[46] O. Räihä, Genetic Synthesis of Software Architecture, University of Tampere, Department of Computer Sciences, Lic. Phil. Thesis, 2008.

[47] J. Kennedy, R.C. Eberhart, Particle swarm optimization, in: Proceedings of the IEEE International Conference on Neural Networks, 1995, pp. 1942–1948.

[48] N. Gold, Hypothesis-based concept assignment to support software maintenance, in: Proceedings of the 22nd International Conference on Software Maintenance, ICSM 01, 2001, pp. 545–548.

[49] C.E. Shannon, The mathematical theory of communications. Bell System, Technical Journal 27 (379–423) (1948) 623–656.

[50] M. Huhns, M. Singh, Service-oriented computing: key concepts and principals, IEEE Internet Computing (2005) 75–81.

[51] G. Canfora, M. Di Penta, R. Esposito, M.L. Villani, (a), An approach for qoS-aware service composition based on genetic algorithms, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, 2005, pp. 1069–1075.

[52] G. Canfora, M. Di Penta, R. Esposito, M.L. Villani, (b) QoS-aware replanning of composite web services, in: Proceedings of IEEE International Conference on Web Services, ICWS'05, 2005, pp. 121–129.

[53] M.C. Jaeger, G. Mühl, QoS-based selection of services: the implementation of a genetic algorithm, in: T. Braun, G. Carle, B. Stiller (Eds.), Kommunikation in Verteilten Systemen (KiVS) 2007 Workshop: Service-Oriented Architectures und Service-Oriented Computing, VDE Verlag, 2007, pp. 359–371.

[54] C. Zhang, S. Su, J. Chen, A novel genetic algorithm for qos-aware web services selection, in: LNCS, vol. 4055, 2006, pp. 224–235.

[55] S. Su, C. Zhang, J. Chen, An improved genetic algorithm for web services selection, in: LNCS, vol. 4531, 2007, pp. 284–295.

[56] L. Cao, M. Li, J. Cao, (a) Cost-driven web service selection using genetic algorithm, in: LNCS, vol. 3828, 2005, pp. 906–915.

[57] L. Cao, J. Cao, M. Li, (b) Genetic algorithm utilized in cost-reduction driven web service selection, in: LNCS, vol. 3802, 2005, pp. 679–686.

[58] R. Garfinkel, G.L. Nemhauser, Integer Programming, John Wiley and Sons, 1972.

[59] G. Canfora, M. Di Penta, R. Esposito, M.L. Villani, A lightweight approach for QoS-aware service composition, in: Proceedings of the ICSOC 2004–short papers. IBM Technical Report, New York, USA, 2004.
[60] S. Wadekar, S. Gokhale, Exploring cost and reliability tradeoffs in architectural alternatives using a genetic algorithm, in: Proceedings of the 10th International Symposium on Software Reliability Engineering, 1999, pp. 104–113.
[61] Y. Che, Z. Wang, X. Li, Optimization parameter selection by means of limited execution and genetic algorithms, in: APPT 2003, in: LNCS, vol. 2834, 2003, pp. 226–235.
[62] T.N. Bui, B.R. Moon, Genetic algorithm and graph partitioning, IEEE Transactions on Computers 45 (7) (1996) 841–855.
[63] S. Shazely, H. Baraka, A. Abdel-Wahab, Solving graph partitioning problem using genetic algorithms, in: Midwest Symposium on Circuits and Systems, 1998, pp. 302–305.
[64] S. Mancoridis, B.S. Mitchell, C. Rorres, Y.-F. Chen, E.R. Gansner, Using automatic clustering to produce high-level system organizations of source code, in: Proceedings of the International Workshop on Program Comprehension, IWPC'98, USA, 1998, pp. 45–53.
[65] D. Doval, S. Mancoridis, B.S. Mitchell, Automatic clustering of software systems using a genetic algorithm, in: Proceedings of the Software Technology and Engineering Practice, 1999, pp. 73–82.
[66] S. Mancoridis, B.S. Mitchell, Y.-F. Chen, E.R. Gansner, Bunch: A clustering tool for the recovery and maintenance of software system structures, in: Proceedings of the IEEE International Conference on Software Maintenance, 1999, pp. 50–59.
[67] B.S. Mitchell, S. Mancoridis, Using heuristic search techniques to extract design abstractions from source code, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'02, USA, July 2002, pp. 1375–1382.
[68] B.S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the Bunch tool, IEEE Transactions on Software Engineering 32 (3) (2006) 193–208.
[69] B.S. Mitchell, S. Mancoridis, On the evaluation of the Bunch search-based software modularization algorithm, Soft Computing 12 (1) (2008) 77–93.
[70] B.S. Mitchell, S. Mancoridis, Modeling the search landscape of metaheuristic software clustering algorithms, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'03, 2003, pp. 2499–2510.
[71] B.S. Mitchell, S. Mancoridis, M. Traverso, Search based reverse engineering, in: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE'02, 2002, pp. 431–438.
[72] K. Mahdavi, M. Harman, R. Hierons, (a) A multiple hill climbing approach to software module clustering, in: Proceedings of ICSM 2003, pp. 315–324.
[73] K. Mahdavi, M. Harman, R. Hierons, (b) Finding building blocks for software clustering, in: LNCS, vol. 2724, 2003, pp. 2513–2514.
[74] M. Harman, R. Hierons, M. Proctor, A new representation and crossover operator for search-based optimization of software modularization, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'02, 2002, pp. 1351–1358.
[75] M. Harman, S. Swift, K. Mahdavi, An empirical study of the robustness of two module clustering fitness functions, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, USA, 2005, pp. 1029–1036.
[76] G. Antoniol, M. Di Penta, M. Neteler, Moving to smaller libraries via clustering and genetic algorithms, in: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, CSMR'03, 2003, pp. 307–316.
[77] M. Di Penta, M. Neteler, G. Antoniol, E. Merlo, A language-independent software renovation framework, The Journal of Systems and Software 77 (2005) 225–240.
[78] S. Huynh, Y. Cai, An Evolutionary approach to software modularity analysis, in: Proceedings of the First international workshop on Assessment of Contemporary Modularization Techniques ACoM'07, ICSE Workshops, 2007, pp. 1–6.
[79] R. Salomon, Short notes on the schema theorem and the building block hypothesis in genetic algorithms, in: Evolutionary Programming VII, in: LNCS, vol. 1447, 1998, pp. 113–122.
[80] A. Tucker, S. Swift, X. Liu, Grouping multivariate time series via correlation, IEEE Transactions on Systems, Man, and Cybernetics. Part B: Cybernetics 31 (2) (2001) 235–245.
[81] L. Kaufman, P. Rousseeuw, Finding Groups in Data: An Introduction to Cluster Analysis, Wiley, 1990.
[82] O. Seng, M. Bauyer, M. Biehl, G. Pache, Search-based improvement of subsystem decomposition, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'05, 2005, pp. 1045–1051.
[83] O. Seng, J. Stammel, D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'06, 2006, pp. 1909–1916.
[84] M. O'Keeffe, M. Ó Cinnéide, Towards automated design improvements through combinatorial optimization, in: Workshop on Directions in Software Engineering Environments (WoDiSEE2004), W2S Workshop -26th International Conference on Software Engineering, 2004, pp. 75–82.
[85] M. O'Keeffe, M. Ó Cinnéide, Search-based software maintenance, in: Proceedings of CSMR'06, 2006, pp. 249–260.
[86] M. O' Keeffe, M. Ó Cinnéide, (a) Search-based refactoring for software maintenance, Journal of Systems and Software 81 (4) (2008) 502–516.
[87] J. Bansiya, C.G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Transactions on Software Engineering 28 (1) (2002) 4–17.
[88] M. O'Keeffe, M. Ó Cinnéide, Getting the most from search-based refactoring, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'07, 2007, pp. 1114–1120.
[89] F. Quaum, R. Heckel, Local search-based refactoring as graph transformation, in: Proceedings of the 1st Symposium on Search-Based Software Engineering, 2009, pp. 43-46.
[90] T. Jiang, N. Gold, M. Harman, L. Zheng, (a) Locating dependence structures using search-based slicing, Information and Software Technology 50 (2008) 1189–1209.
[91] T. Jiang, M. Harman, Y. Hassoun, (b) Analysis of Procedure Splittability, in: Proceedings of the 15th Workshop on Reverse Engineering, 2008, pp. 247–256.
[92] D. Fatiregun, M. Harman, R. Hierons, Evolving transformation sequences using genetic algorithms, in: Proceedings of the 4th International Workshop on Source Code Analysis and Manipulation, SCAM 04, 2004, pp. 65–74.
[93] K. Williams, Evolutionary algorithms for automatic parallelization, Ph.D. Thesis, Department of Computer Science, University of Reading, UK. 1998.
[94] C. Ryan, L. Ivan, Automatic parallelization of arbitrary programs, in: Proceedings of EUROGP'99, in: LNCS, vol. 1598, 1999, pp. 244–254.
[95] M. Harman, L. Tratt, Pareto optimal search based refactoring at the design level, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'07, 2007, pp. 1106–1113.
[96] E. Falkenaur, Genetic Algorithms and grouping problems, Wiley, 1998.

[97] L. Briand, J. Wüst, J. Daly, V. Porter, Exploring the relationships between design measures and software quality in object oriented systems, Journal of Systems and Software 51 (2000) 245–273.
[98] M. O' Keeffe, M. Ó Cinnéide, (b) Search-based refactoring: an empirical study, Journal of Software Maintenance and Evolution: Research and Practice 20 (2008) 345–364.
[99] M. Dorigo, Optimization, Learning and Natural Algorithms, Ph.D. thesis, Dipartimento di Elettronica, Politecnico di Milano, 1992.
[100] S. Horwitz, T. Reps, D.W. Binkley, Interprocedural slicing using dependence graphs, in: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, 1988, pp. 35–46.
[101] H. Naeimi, A. DeHon, A greedy algorithm for tolerating defective crosspoints in NanoPLA design, in: Proceedings of the International Conference on Field-Programmable Technology, ICFPT2004, 2004, pp. 49–56.
[102] H.A. Sahraoui, R. Godin, T. Miceli, Can metrics help bridging the gap between the improvement of OO design quality and its automation? in: Proceedings of the International Conference on Software Maintenance, ICSM'00, 2000, pp. 154–162.
[103] T. Mens, S. Demeyer, Future trends in evolution metrics, in: Proceeding of the International. Workshop on Principles of Software Evolution, 2001, pp. 83–86.
[104] M. Harman, J. Clark, Metrics are fitness functions too, in: 10th International Software Metrics Symposium, METRICS 2004, USA 2004, pp. 58–69.
[105] S. Bouktif, B. Kégl, H. Sahraoui, Combining software quality predictive models: an evolutionary approach, in: Proceedings of the International Conference on Software Maintenance, ICSM'02, 2002, pp. 385–392.
[106] S. Bouktif, D. Azar, H. Sahraoui, B. Kégl, D. Precup, Improving rule set based software quality prediction: a genetic algorithm-based approach, Journal of Object Technology 3 (4) (2004) 227–241.
[107] S. Bouktif, H. Sahraoui, G. Antoniol, Simulated annealing for improving software quality prediction, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'06, 2006, pp. 1893–1900.
[108] R.A. Vivanco, D. Jin, Selecting object-oriented source code metrics to improve predictive models using a parallel genetic algorithm, in: Proceedings of OOPSLA'07, 2007, pp. 769–770.
[109] W.J. Youden, How to evaluate accuracy, in: Materials Research and Standards, ASTM, 1961.
[110] W. Miller, D.L. Spooner, Automatic generation of floating-point test data, IEEE Transactions on Software Engineering 2 (3) (1976) 223–226.
[111] K.F. Fischer, A test case selection method for the validation of software maintenance modifications, in: Proceedings of International Computer Software and Applications Conference, COMPSAC'77, 1977, pp. 421–426.
[112] K.F. Fischer, F. Raji, A. Chruscicki, A methodology for retesting modified software, in: Proceedings of National Telecommuncations Conference, NTC'81, 1981, pp. 1–6.
[113] S. Kirkpatrick, C. Gelatt, M. Vecchi, Optimization by simulated annealing, Science 220 (1983) 671–680.
[114] N. Cramer, A representation for the adaptive generation of simple sequential programs, in: Proceedings of the International Conference on Genetic Algorithms and their Applications, Carnegie-Mellon University, pp. 183–187.
[115] F. Glover, Future paths for Integer Programming and Links to Artificial Intelligence, Computers and Operations Research 5 (1986) 533–549.
[116] J. Hartmann, D.J. Robson, Revalidation during the software maintenance phase, in: Proceedings of the 1989 Conference on Software Maintenance, EEE Press, 1981, pp. 70–80.
[117] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, S. Karapoulis, Application of genetic algorithm to software testing, in: Proceedings of the 5th International Conference on Software Engineering and Applications, 1992, pp. 625–636.
[118] M. Schoehauer, S. Xanthakis, Constrained GA optimization, in: Proceedings of the 5th International Conference on Genetic Algorithms, ICGA'93, 1993, pp. 573–580.
[119] C. Chao, J. Komada, Q. Liu, M. Muteja, Y. Alsalgan, C. Chang, An application of genetic algorithms to software project management, in: Proceedings of the 9th International Conference on Advanced Science and Technology, 1993, pp. 247–252.
[120] M. Pei, E.D. Goodman, Z. Gao, K. Zhong, Automated software test data generation using a genetic algorithm, 1994, available at: www.egr.msu.edu/~pei/paper/GApaper94-02.ps.
[121] T. Minohara, Y. Tohma, Parameter estimation of hypergeometric distribution software reliability growth model by genetic algorithms, in: Proceedings of the 6th Symposium on Software Reliability Engineering, 1995, pp. 324–329.
[122] R. Ferguson, B. Korel, Software test data generation using the chaining approach, in: Proceedings of the IEEE International Test Conference on Driving Down the Cost of Test, 1995, pp. 703–709.
[123] B. Jones, H-H. Sthamer, D.E. Eyres, Automatic structural testing using genetic algorithms, Software Engineering Journal 11 (5) (1996) 299–306.
[124] E. Alba, J.M. Troya, Genetic algorithms for protocol validation, in: Proceedings of the 4th International Conference on Parallel Problem Solving from Nature, PPSN'96, 1996, pp. 870–879.
[125] J.T. Alander, T. Mantere, G. Moghadampour, Testing software response times using a genetic algorithm, in: Proceedings of the 3rd Nordic Workshop on Genetic Algorithms and their Applications, 3NWGA, 1997, pp. 293–298.
[126] M.C. Sinclair, S.H. Shami, Evolving simple software agents: comparing genetic algorithm and genetic progrraming performance, in: Proceedings of the 2nd International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA'97, 1997, pp. 421–426.
[127] R. Feldt, Generating multiple diverse software versions with genetic programming, in: Proceedings of the 24th EUROMICRO Conference, 1998, pp. 387–394.
[128] N. Tracey, J. Clark, K. Mander, Automated program flaw findign using simulated annealing, in: Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'98, 1998, pp. 73–81.
[129] J.J. Dolado, L. Fernandez, Genetic programming, neural networks and linear programming in software project estimation, in: Proceedings of International Conference on Software Process Improvement, Research, Education and Training, INSPIRE III, 1998, pp. 157–171.
[130] A. Nisbet, GAPS: a compiler framework for genetic algorithm (GA) optimised parallelisation, in: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, HPCN'98, 1998, pp. 987–989.
[131] Y. Monnier, J-P. Beauvais, A.-M. Déplanche, A genetic algorithm for, scheduling tasks in real-time distributed system, in: Proceedings of the 24th EUROMICRO Conference, EUROMICRO'98, 1998, pp. 20708–20714.
[132] C. Ryan, Automatic Re-engineering of Software Using Genetic Programming, Kluwer Academic Publishers, 1999.
[133] M.P. Evett, T.M. Khoshgoftaar, P-D. Chien, E.B. Allen, Using genetic programming to deterine software quality, in: Proceedings of the 12th International Floridy Artificial Intelligence Research Society Conference, FLAIRS'99, 1999, pp. 113–117.

[134] K.D. Cooper, P.J. Schielke, D. Subramanian, Optimizing for reduced code space using genetic algorithms, in: Proceedings of the ACM SIGPLAN 1999 Woskhop on Languages, Compilers and Tools for Embedded Systems, LCTES'99, 1999, pp. 1–9.
[135] N. Mansour, K. El-Fakih, Simulated annealing and genetic algorithms for optimal regression testing, Journal of Software Maintenance: Research and Practice 11 (1) (1999) 19–34.
[136] K. El-Fakih, H. Yamaguchi, G.V. Bochmann, A method and a genetic algorithm for deriving protocols for distributed applications with minimum communication cost, in: Proceedings of the 11th International Conference on Parallel and Distributed Computing and Systems, PDCS'99, 1999, pp. 863–868.
[137] J.A. Clark, J.J. Jacob, Searching for a solution: engineering tradeoffs and the evolution of provably secure protocols, in: Proceedings of the 2000 IEEE Symposium on Security and Privacy, 2000, pp. 82–95.
[138] A.J. Bagnall, V.J. Rayward-Smith, I.M. Whittley, The next release problem, Information and Software Technology 43 (14) (2001) 883–890.
[139] B. Mitchell, A Heuristic Search Approach to Solving the Software Clustering Problem, Ph. D. Thesis, Drexel University, Philadelphia, 2002.
[140] Y. Shan, R.I. McKay, C.J. Lokan, D.L. Essam, Software project effort estimation using genetic programming, in: Proceedings of the 2002 IEEE International Conference on Communications, Circuits and Systems and West Sino Expositions, 2002, pp. 1108–1112.
[141] D. Fatiregun, M. Harman, R. Hierons, Search based transformations, in: Proceedings of the 2003 Conference on Genetic and Evolutionary Computation, GECCO'03, 2003, pp. 2511–2512.
[142] M.B. Cohen, C.J. Colbourn, A.C.H. Ling, Augmenting simulated annealing to build interaction test suites, in: Proceedings of the 14th International Symposium on Software Reliability Engineering, 2003, pp. 394–405.
[143] B.S. Mitchell, S. Mancoridis, M. Traverso, Using interconnection style rules to infer software architecture relations, in: Proceedings of the 2004 Conference Genetic and Evolutionary Computation, GECCO'04, 2004, pp. 1375–1387.
[144] G. Antoniol, M. Di Penta, M. Harman, Search-based techniques for optimizing software project resource allocation, in: Proceedings of the 2004 Conference on Genetic and Evolutionary Computation, GECCO'04, 2004, pp. 1425–1426.
[145] D. Fatiregun, M. Harman, R. Hierons, Search-based amorphous slicing, in: Proceedings of the 124th International Working Conference on Reverse Engineering, WCRE'05, 2005, pp. 3–12.
[146] H. Li, C.P. Lam, An ant colony optimization approach to test sequence generation for statebased software testing, in: Proceedings of the 5th International Conference on Quality Software, QSIC'05, 2005, pp. 255–264.
[147] L. Yang, B.F. Jones, S.-H. Yang, Genetic algorithm based software integration with minimum software risk, Information and Software Technology 48 (3) (2006) 133–141.
[148] Y. Zhang, M. Harman, S.A. Mansouri, The multi-objective next release problem, in: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, GECCO'07, 2007, pp. 1129–1137.
[149] E. Alba, F. Chicano, Ant colony optimization for model checking, in: Proceedings of the 11th International Conference on Computer Aided Systems Theory, EUROCAST 2007, 2007, pp. 523–530.
[150] C. Johnson, Genetic programming with fitness based on model checking, in: Proceedings of the 10th European Conference on Genetic Programming, 2007, pp. 114–124.
[151] S. Yoo, M. Harman, Pareto efficient multi-objective test case selection, in: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA'07, 2007, pp. 140–150.
[152] R. Lange, S. Mancoridis, Using code metric histograms and genetic algorithms to perform author identification for software forensics, in: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, GECCO'07, 2007, pp. 2082–2089.
[153] W. Shyang, C. Lakos, Z. Michalewicz, S. Schellenberg, Experiments in applying evolutionary algorithms to software verification, in: Proceedings of the IEEE Congress on Evolutionary Computation, CEC'08, 2008, pp. 3531–3536.
[154] A. Arcuri, X. Yao, A novel co-evolutionary approach to automatic software bug fixing, in: Proceedings of the IEEE Congress on Evolutionary Computation, CEC'08, 2008, pp. 162–168.
[155] Y. Zhang, A. Finkelstein, M. Harman, Search based requirements optimisation: existing work & challenges, in: Proceedings of the 14th International Working Conference, Requirements Engineering: Foundation for Software Quality, REFSQ'08, 2008, pp. 88–94.
[156] E. Díaz, J. Tuya, R. Blanco, J. Dolado, A tabu search algorithm for structural testing, Computers & Operations Research 35 (10) (2008) 3052–3072.
[157] A.J. Ramirez, D.B. Knoester, B.H.C. Cheng, P.K. McKinley, Applying genetic algorithms to decision making in autonomic computing systems, in: Proceedings of the 6th International Conference on Autonomic Computing, ICAC'09, 2009, pp. 97–106.

Paper VII

Outi Räihä, Kai Koskimies and Erkki Mäkinen, Complementary Crossover for Genetic Software Architecture Synthesis, In: *Proc. of the International Conference on Intelligent Systems Design and Application (ISDA'10)*, Cairo, Egypt. December 2010, IEEE Press, 260-265.

© 2010 IEEE. Reprinted, with permission

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University'of Tampere's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this material, you agree to all provisions of the copyright laws protecting it.

Complementary Crossover for Genetic Software Architecture Synthesis

Outti Räihä, Kai Koskimies
Department of Software Systems
Tampere University of Technology,
Tampere, Finland
{outi.raiha, kai.koskimies}@tut.fi

Erkki Mäkinen
Department of Computer Sciences
University of Tampere
Tampere, Finland
em@cs.uta.fi

***Abstract*— Techniques exist to synthesize software architecture using genetic algorithms that employ transformations based on mutations and crossover. In this paper, we demonstrate that complementary crossover can significantly improve this technique. We study two versions of complementary crossover, one in which parents are selected so that they complement each other but the genes are inherited randomly from the parents, and another in which the genes are inherited in a more purposeful way. Empirical studies on two sample systems suggest that the complementary crossover outperforms the traditional crossover in genetic software architecture synthesis especially in the presence of mutations that provide delayed reward.**

Keywords: complementary crossover, search-based software engineering, software design, software architecture

I. INTRODUCTION

Design of software architecture is one of the most critical and intellectually demanding activities of software engineering. In our previous work [11, 12] we have taken the viewpoint that software architecture essentially consists of applications of known good practices and solutions of architectural design in the context of a particular application. Assuming that both the architecture and the general solutions can be formally represented, and that the quality of an architecture can be effectively measured, the problem of architecture design becomes a search problem that can be solved by a heuristic search algorithm.

We have shown [11, 12] that genetic algorithms (GAs) [9] can be applied to the problem of architectural design, interpreting general solutions (such as architectural styles [14] and design patterns [6]) as mutations and using various architectural metrics as a basis for the fitness function. In addition to mutation, changes were induced by crossover, where two parent architectures convey part of their structure to the offspring. The probability of becoming a parent was higher for "good" architectures, but otherwise the crossover was realized in a random manner.

However, following the so-called genetic compatibility hypothesis in biology [18], the idea of purposeful parent selection has been proposed for GAs (e.g., by Fernandes and Rosa [5]). Basically, the idea is to select parents in such a way that they complement each other, thus producing more likely desirable qualities in offspring. We will use the term complementary crossover to refer to such techniques.

Complementary crossover requires some method to assess the matching of parents. For software architecture synthesis, this idea is particularly amenable, because software architectures are evaluated in terms of few quality attributes that can be measured separately. For example, it seems attractive to select one parent with good modifiability characteristics and the other with good efficiency characteristics, in the hope that the offspring could inherit, at least to some extent, both desirable quality attributes. Intuitively, this could speed up the evolution and produce more balanced solutions when compared to random crossover with randomly chosen parents, as modifiability and efficiency are exceptionally difficult to optimize simultaneously.

In this paper we explore the potential of complementary crossover in the genetic synthesis of software architecture. We will study first the case where the parents are complementary but the inherited parts of the parents are selected randomly, and then the case where the inherited parts are selected purposefully so that the intended characteristics are more likely transferred to the offspring. We experiment with our new crossover operators using two sample systems of different character.

II. RELATED WORK

Recently, approaches dealing with high level structures, such as design patterns [1] and architectures [15] have gained more interest in search-based software engineering [8, 10]. We will here limit the discussion to applications of GAs where the crossover is given a special role.

Harman et al. [7] study a new crossover operator in the area of clustering. The crossover attempts to conserve building blocks by ensuring that at least one complete cluster from one of the parents is kept intact in the crossover process. This is done by directly copying a complete cluster from one parent in the beginning of the crossover parent.

Fernandes and Rosa [5] propose a reproduction method, where the roulette wheel selection is used to choose one parent and some subgroup of the population. The other parent is then the one in the selected subgroup that differs most from the parent chosen first. Hamming distance is used to calculate the difference, and the method is argued to retain building blocks.

Dolin et al. [4] base the choosing of parents on "fitness cases". One of the parents is chosen in a standard fashion,

978-1-4244-8136-1/10/$26.00 ©2010 IEEE

while the other one is chosen so that it performs well in the fitness cases where the other one doesn't. It is hoped that the two parents thus complete each other and maximize the fitness for the "ultimate" offspring.

Wildman and Parks [17] compare different breeding strategies for multi-objective GAs. They study crossovers based on genotypic, phenotypic and ranking dissimilarities and pairing restrictions. They conclude that pairing strategies that combine dissimilar parents produce better results.

Räihä et al. [13] study the effect of crossover in genetic architecture synthesis. Results showed that asexual reproduction performed better than the traditional random crossover. However, asexual mutation resulted in very homogenous populations and seemed to land on a local optimum very early. To this end, the present paper aims at a more sophisticated crossover, finding solutions with higher quality than the asexual method.

III. GENETIC SYNTHESIS OF SOFTWARE ARCHITECTURE

In this section we describe our approach to automate software architecture design by using GAs, to synthesize software architectures. We assume that the reader is familiar with the basics of GAs, as given, e.g., by Michalewicz [9]. In what follows, we give an encoding of possible solutions, an initial population, mutation and crossover operators, a fitness function and a selection operator for choosing the survivors for the next generation.

A. Requirements

For expressing functional requirements we identify and express the primary use cases of the system, and refine them into sequence diagrams depicting the interaction between major components required to accomplish the use cases. This is a manual task, as the major components have to be decided, typically based on domain analysis.

In our approach, a so-called null architecture represents a basic functional decomposition of the system, given as a UML class diagram. The null architecture can be mechanically derived from the use case sequence diagrams: the (classes of the) participants in the sequence diagram become the classes, the operations of a class are the incoming call messages of the participants of that class, and the dependency relationships between the classes are inferred from the call relationships of the participants. Additionally, if a component has a significant state or it manages a significant data entity (e.g., a data base), it will become an attribute in the class.

We have used here two sample systems: an e-home control system (called hereafter ehome), which represents a typical embedded system, and a robot war game application (robowar), which represents a desktop system. Ehome controls various devices, providing an interface to allow the user to manage the home. The ehome system requirements lead to 56 operations and 90 dependencies between the operations. Robowar is a computer game where robots with different characteristics fight against each other. Robowar requirements lead to 57 operations and 73 dependencies between them. The null architecture for ehome contains 12 classes and the null architecture for robowar contains 22 classes.

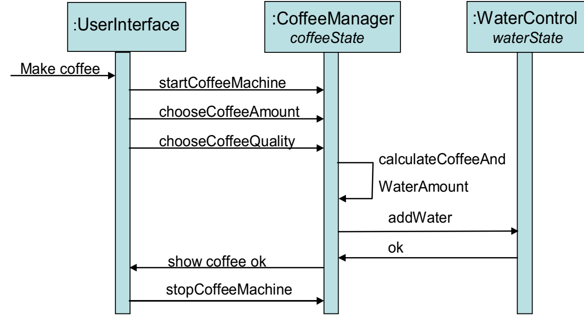


Figure 1. Make coffee use case refined

Use cases for the ehome are assumed to consist of logging in, changing the room temperature or its unit, making coffee, moving drapes, and playing music. In Fig. 1, the coffee making use case has been refined into a sequence diagram. A fragment of the null architecture for ehome is given in Figure 2, representing the same part of the system as Figure 1.

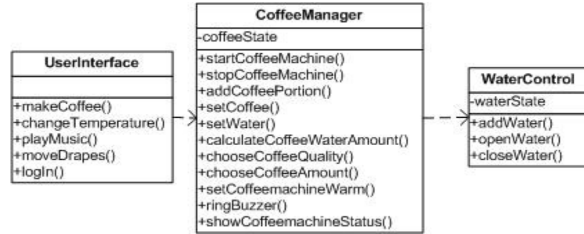


Figure 2. A fragment of the null architecture for ehome

B. Genetic Representation

When the architectural data is encoded into a chromosome form, two kinds of data are given regarding each operation. Firstly, the basic information contains the operations depending on it, its name, type, frequency of use, parameter size, and sensitiveness to variation. Secondly, there is the information regarding the operation's place in the architecture: the class(es) it belongs to, the interface it implements, the message dispatcher it uses (further explained in Section III.C), the operations that call it through the message dispatcher, the design patterns it is a part of and the class it is assigned to in the null architecture. The message dispatcher is given a separate field as opposed to other patterns for efficiency reasons. All data regarding an operation is encoded as a *supergene* [1].

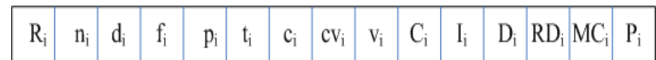


Figure 3. Supergene for responsibility r_i

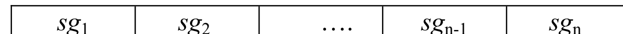


Figure 4. Chromosome for a system with *n* responsibilities

A supergene is depicted in Figure 3. The chromosome handled by the genetic algorithm is gained by collecting the

supergenes, i.e., all data regarding all responsibilities, thus achieving a whole view of the functional requirements for the architecture. Figure 4 illustrates the chromosome structure.

The initial population is made by first creating the desired number of individuals with the null architecture. A random pattern is then inserted into each individual, as a population should not consist entirely of clones. In addition, a special individual with no initial patterns is left in the population.

C. *Mutation Operations*

The actual architectural design means here the application of various standard architectural solutions called collectively patterns: the result of genetic architecture synthesis is the null architecture augmented with patterns. The patterns have been chosen to represent solutions on different levels: high-level architectural styles [14] (message dispatcher and client-server), medium-level design patterns [6] (Façade and Mediator), and low-level design patterns [6] (Strategy, Adapter and Template Method). The mutations are implemented in pairs of introducing a specific pattern or removing it. The dispatcher architecture style makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the operations can communicate through it.

The actual mutation probabilities are given as input. Selecting the mutation is made with a roulette wheel selection [9]. Null mutation and crossover are also included in the wheel. The standard crossover is implemented as a traditional one-point crossover where the crossover point is selected randomly. Each individual has a chance of reproducing in each generation: if the first roulette selection lands on a mutation, another selection is performed after the mutation has been administered. If the second selection lands on the crossover slice, the individual may produce offspring. In any other case, the second selection is not taken into account.

Additionally, a corrective operation is performed to ensure that the architecture stays coherent. The operation ensures that the patterns present in the system stay coherent and "legal", and checks that no anomalies are brought to the design, such as interfaces without any users or tasks implementing more than one interface.

D. *Fitness Function and Selection*

The fitness function is based on widely used software metrics [2]. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-sever architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. A complexity metric is added to penalize having many classes and interfaces.

Dividing the fitness function into sub-functions answers the demands of the real world. By separating efficiency and modifiability, which are especially difficult to optimize simultaneously, we can assign a bigger weight to the more desired quality aspect. When w_i is the weight for the respective sub-fitness sf_i, the fitness function $f_c(x)$ for chromosome x can be expressed as

$$f_c(x) = w_1*sf_1 - w_2*sf_2 + w_3*sf_3 - w_4*sf_4 - w_5*sf_5.$$

Here, sf_1 measures positive modifiability, which takes into account how well interfaces are used. In addition, calls between operations that are handled via message dispatcher are rewarded, and the sensitiveness to variation of the operations is used to enhance the reward. The message dispatcher is considered to have exceptional potential in increasing the independency of different component, and is thus given a very high reward, which has the drawback of potentially dominating the fitness value. Negative modifiability is calculated in sf_2 by penalizing direct calls between classes.

As for efficiency, sf_3 measures positive efficiency by rewarding structures which lead to minimal amount of calls between different classes as well as calls between operations within the same class. The operation's required amount of data is also considered and used to increase the reward. Negative efficiency (sf_4) in turn counts the relation of calls between classes and within classes, and the amount of calls to the message dispatcher and through servers, which are especially penalized by taking into account the frequency of calls to the operations involved.

Finally, complexity is penalized in sf_5 by calculating the amount of classes and interfaces.

Selecting the individuals for each generation is also made with the roulette wheel method. Here the size of each slice is based on the rank of an individual. This is combined with elitism to ensure that the very top of each population is kept for the next generation.

IV. COMPLEMENTARY CROSSOVER

In this paper, we compare the standard crossover to *complementary crossover* and *complementary gene-selective crossover*. The complementary crossover attempts to combine parents so that they represent different fitness aspects. In the beginning of a generation cycle, all individuals are sorted in ascending order based on their modifiability and efficiency fitness values. An individual's ranks in the lists are referred to as its modifiability and efficiency rank, respectively. If an individual's modifiability rank is higher than its efficiency rank, then the individual is placed in the mother pool. Otherwise, it is placed in the father pool. Thus, the mothers represent modifiable individuals, while the fathers represent efficient individuals.

The individuals picked for reproduction are chosen in the standard roulette wheel selection. Once the parents are selected, they are coupled based on the pool division. Only mother-father pairs are allowed in crossover; if there are more of one or the other, they are simply left as they are in the population. In this case, the crossover point is still chosen randomly, and two offspring are produced, as with the standard crossover.

The complementary gene-selective crossover uses the same parent selection process as the simple complementary crossover, but further attempts to take advantage of the different properties of the parents by searching for an optimal crossover point. This is done by searching for the

optimal blocks in the parents: the most modifiable piece of architecture in the mother, and the most efficient section of the father.

The blocks are located by using the maximum contiguous subsequence sum algorithm [16]. Each supergene is assigned its individual fitness value for modifiability and efficiency. This value can be calculated by the change it causes in the fitness values. Two integer vectors, one for modifiability and one for efficiency, are achieved, where the integer value in index i is the respective quality value for supergene i. The maximum contiguous subsequence sum is then calculated for both vectors, and the first and last indexes for the subsequence are recorded.

In order to keep intact both the modifiability block provided by the mother and the efficiency block provided by the father, the crossover point cp needs to be selected so that it lands between the blocks. Thus, if the modifiability block begins at index l and ends at index k, and the efficiency block begins at index m and ends at index p, where $k < m$ and $p < n$, n being the number of operations, cp is selected randomly so that $k < cp < m$. The complementary, gene-selective crossover is illustrated in Figure 5, where two parents are combined to create an offspring with the best blocks from each. Note that for the complementary gene-selective crossover only one offspring is produced, as there is no sense to create one optimal offspring and one "leftover" offspring.

Also, if the blocks of the initial mother-father pair overlap, another father is selected from those selected for crossover until either a suitable pair for the current mother has been found or there are no more fathers selected for crossover, in which case the mother is left as it was in the population and does not participate in crossover.

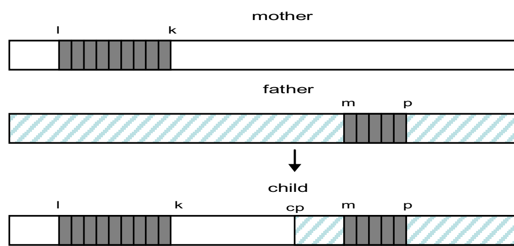


Figure 5. Complementary gene-selective crossover

V. EXPERIMENTS

We used two sample systems, the robowar and the ehome to see how the different approaches to crossover affect the development of the fitness curve. In our experiments we used a population of 100 individuals and 250 and 750 generations. The curves are averages of 20 test runs. The y-value of the curve represents the average fitness of the 10 best individuals in each generation. For all tests, the probability for crossover was set to 4%. The weights for all sub-fitnesses were set to 1.

In Figures 6-9 the fitness values obtained in our tests are depicted in logarithmic scale. This is due to the high fitness values caused by the heavy use of message dispatcher, as mentioned in Section III. The logarithmic scale requires that the fitness values must be non-negative. However, this is not always the case if the function is defined as explained in Section III. To that end, we have added a constant (1243, to be exact) to all our fitness values before drawing the curves in Figures 6-9.

Figure 6 depicts the fitness curves for the different crossovers for ehome. As can be seen, the standard crossover curve remains very stable as the growth is so small it does not show on logarithmic scale (change in actual fitness values is about 250 "fitness units"). The complementary crossovers behave very differently, as they first begin descending, but once they start to actually ascend, they achieve much higher values than the standard crossover. At this point, merely selecting parents would appear to achieve better value than if also the crossover point is selected.

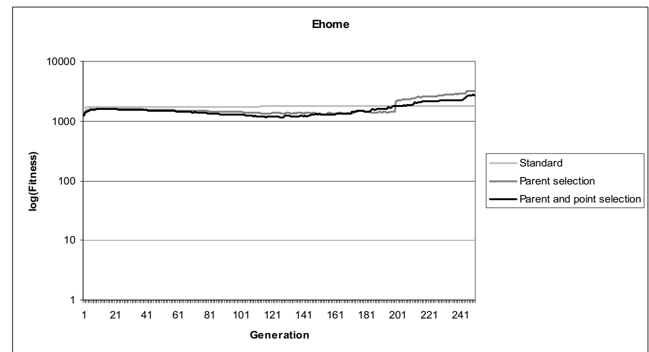


Figure 6. Fitness curves for different crossovers and 250 generations for ehome

The rather odd slump in the development of the complementary crossover curves is interesting, as weaker individuals seem to appear and even more curiously survive in the population. This can be explained by the relation of the different quality attributes, and we will discuss it further in Section VI.

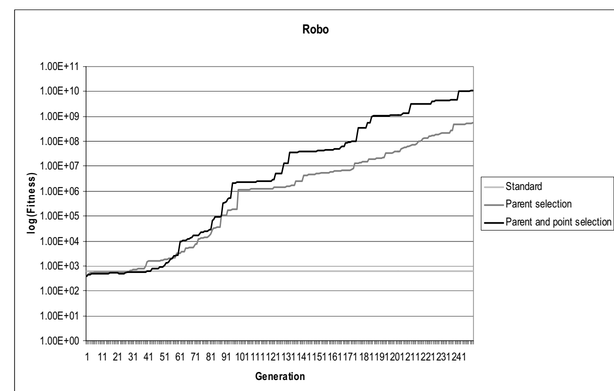


Figure 7. Fitness curves for different crossovers and 250 generations for robowar

Figure 7 shows the respective fitness curves for the robowar system. In this case the difference between the

standard and complementary crossovers is drastic: the fitness curve for the standard crossover develops minimally, while the complementary crossover curves express exponential growth. Here the most refined crossover, where also the crossover point is selected, already dominates after 250 generations.

Especially because of the curves for ehome, where there was minimal development compared to the robowar curves, we also wanted to see what kind of results could be achieved with longer runs, and ran the same tests, only now increasing the number of generations to 750. Figure 8 shows the fitness curves for ehome with 750 generations. The most advanced crossover, complementing parents with gene selection, has now risen to the top, and the complementary crossovers develop steadily, while the fitness curve for the standard crossover shows very little development.

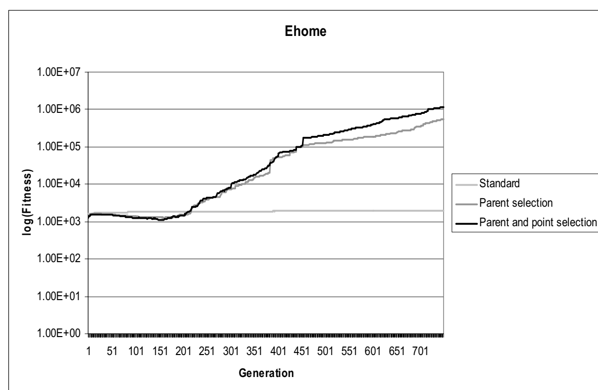


Figure 8. Fitness curves for different crossovers and 750 generations for ehome

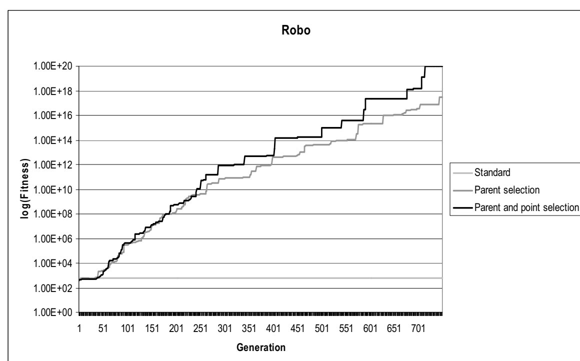


Figure 9. Fitness curves for different crossovers and 750 generations for robowar

Figure 9 shows the fitness curves for robowar with 750 generations. The trend here is the same as with 250 generations: the highest fitness values are achieved when both parents and the crossover point are selected, i.e., with the complementary gene-selective crossover. Here the exponential shape of the curve that could already be seen within 250 generations is even clearer. This can be explained partially by the structure of the robowar system. Compared to ehome, it contains smaller classes and more connections between classes. Thus, there are more possibilities to add modifiability increasing patterns between classes and dispatcher connections between responsibilities. This, in turn. results in more building blocks, and especially the modifiability block can grow extremely large. The standard crossover fitness curve does develop too, but the change is so minimal that it does not show on the logarithmic scale.

VI. DISCUSSION

In addition to fitness curves, the actual architecture proposals should be evaluated in order to determine the complete effect of the presented crossover operations. The most visible effect is the appearance of the message dispatcher architecture style. With the standard crossover (250 generations), there were no solutions for the ehome system where the message dispatcher was present. When the parents were selected, the message dispatcher was present in 18 of 20 cases, and when also the crossover point was selected, the message dispatcher appeared in all of the solutions. For the robowar system, the message dispatcher was present in none of the solutions achieved with the standard crossover, while with both versions of complementary crossover it was present in all of the solutions (after 250 generations). With the longer runs (750 generations), the dispatcher is present in all cases for both systems when the either complementary crossover is used, but in none of the cases for the standard crossover.

Thus, it seems that the new crossovers enable more controversary mutations to appear and survive throughout the generations. Even solutions that temporarily weaken the result are accepted. Initially, there are no or very few patterns in the architecture, and thus it is as efficient as possible, considering the null architecture (optimal efficiency would be achieved by placing all responsibilities in the same class). Thus, applying mutations increases modifiability and decreases efficiency. The complementary crossover enables more drastic changes, where initially the negative effect in efficiency and complexity is larger than the positive effect on modifiability. Thus the overall result is that the total fitness decreases.

The message dispatcher is a perfect example where with only a few connections, the penalty in terms of efficiency is much bigger than the reward in terms of modifiability. However, when the parents are chosen from different pools, the mothers are most likely individuals where the dispatcher is present, as it has the biggest single effect on the modifiability fitness. Furthermore, the “modifiability block” found in gene-selective crossover is also most likely to be a series of operations that communicate through the message dispatcher.

Figure 10 depicts an example solution architecture for ehome, achieved using the complementary gene-selective crossover. For clarity, the patterns have been drawn using a shorthand notation, with dashed boxes. This example shows how communication between classes is centered to message dispatcher, and very few classes communicate directly. This

is especially beneficial, when classes operate as individual units, and direct communication is undesired.

Several Template Method, Strategy and Adapter patterns can also be seen in the proposed solution. For example, the Template Method related to the Coffee Machine (extracting the showCoffeeMachineStatus operation) seems like a particularly wise solution, as the operation has a high variability level, and the implementation is very likely to be changed if the system is updated.

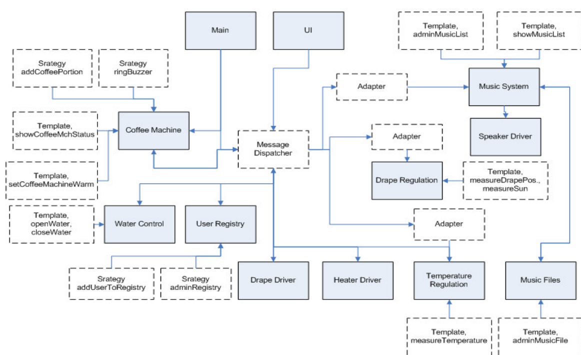


Figure 10. Example solution for ehome

VII. Conclusions

We have presented experimental results achieved with complementary (gene-selective) crossover, applied to software architecture design. We hypothesized that it would be beneficial to take into account the strengths of different individuals in crossover, and to try to combine the strengths of the parents. Our experiments suggest that the complementary crossover and its more refined, gene-selective version both provide more versatility in the produced architectures and enable more complex solutions, leading to significantly better fitness averages.

Thus, it seems that the complementary crossover should be preferred for genetically synthesizing software architectures, as many design choices do not provide instant reward, and allowing a momentary weakening in fitness value will result in better results over a longer period of time. However, the complementary crossover has its weaknesses: it favored "critical" mutations (message dispatcher), which are desirable in certain types of systems but less desirable in others. This suggests that more tests are required on different kinds of example systems and the crossover and its interplay with the fitness function should be further studied.

Finally, it should be emphasized that since the architecture proposals produced by a GA always carry a random element, it is advisable to produce a (small) set of candidate proposals which are subject to human selection. We are currently studying the construction of a Pareto front [3] of such proposals with respect to the fitness metrics used.

Acknowledgment

This work has been funded by the Academy of Finland (project Darwin). The authors wish to thank Prof. Hanna Kokko for useful discussions on biological evolution.

References

[1] M. Amoui, S. Mirarab, S. Ansari and C. Lucas, A genetic algorithm approach to design evolution using design pattern transformation. *International Journal of Information Technology and Intelligent Computing* **1**, 2006, pp. 235-245.
[2] S.R. Chidamber and C.F. Kemerer, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* **20** (6), 1994, pp. 476-492.
[3] K. Deb, Evolutionary algorithms for multicriterion optimization in engineering desing, In: *Proc. of EUROGEN'99*, 1999, pp. 135-161.
[4] B. Dolin, M. G. Arenas and J. J. Merelo, Opposites attract: complementary phenotype selection for crossover in genetic programming. In: *Proc. of PPSN'02*, LNCS **2439**, Springer, 2002, pp. 142-152.
[5] C. Fernandes and A. Rosa, A study on non-random mating and varying population size, In: *Proc. of CEC'01*, 2001, pp. 60-66.
[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.
[7] M. Harman, R. Hierons and M. Proctor, A new representation and crossover operator for search-based optimization of software modularization. In: *Proc. of GECCO'02*, 2002, 1351-1358.
[8] M. Harman, S.A. Mansouri, Y. Zhang, Search based software engineering: a comprehensive review of trends, techniques and applications. Technical report TR-09-03, King's College, London, 2009.
[9] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolutionary Programs.* Springer-Verlag, 1992.
[10] O. Räihä, A survey on search-based software design, *Computer Science Review*, 2010. In press. http://dx.doi.org/10.1016/j.cosrev.2010.06.001
[11] O. Räihä, K. Koskimies and E. Mäkinen, Genetic synthesis of software architecture. In: *Proc. of SEAL'08*, LNCS **5361**, Springer, 2008, pp. 565-574.
[12] O. Räihä, K. Koskimies and E. Mäkinen, Scenario-based genetic synthesis of software architecture. In: *Proc. of ICSEA'09*, 2009, pp. 437-445.
[13] O. Räihä, K. Koskimies and E. Mäkinen, Empirical study on the effect of crossover in genetic software architecture synthesis. In: *Proc. of NaBIC'09*, 2009, pp. 619-625.
[14] M. Shaw and D. Garlan, *Software Architecture – Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
[15] C.L. Simons and I.C. Parmee, A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design. *Engineering Optimization* **39** (5) 2007, pp. 631-648.
[16] M. A. Weiss, *Data Structures and Problem Solving Using Java*. Addison-Wesley, 1998.
[17] A. Wildman and G. Parks, A comparative study of selective breeding strategies in a multiobjective genetic algorithms, *Proc. EMO'03*, LNCS **2632**, 2003, pp. 418-432.
[18] J.A. Zeh and D.W. Zeh, The evolution of polyandry. 1. Intragenomic conflict and genetic incompatibility. *Proc. Roy. Soc. Lond. B* **263,** 1996, pp.1711-1717.

Paper VIII

Outi Räihä, Hadaytullah, Kai Koskimies and Erkki Mäkinen, Synthesizing Architecture from Requirements: A Genetic Approach, In: P. Avgeriou, J. Grundy, J. G. Hall, P. Lago and I. Mistrik (eds), *Relating Software Requirements and Architectures*, Springer, to appear.

© Springer-Verlag Berlin Heidelberg 2011. This is reprinted with permission.

Chapter 18
Synthesizing Architecture from Requirements: A Genetic Approach

Outi Räihä, Hadaytullah Kundi, Kai Koskimies, and Erkki Mäkinen

Abstract The generation of software architecture using genetic algorithms is studied with architectural styles and patterns as mutations. The main input for the genetic algorithm is a rudimentary architecture representing the functional decomposition of the system, obtained as a refinement of use cases. Using a fitness function tuned for desired weights of simplicity, efficiency and modifiability, the technique produces a proposal for the software architecture of the target system, with applications of architectural styles and patterns. The quality of the produced architectures is studied empirically by comparing these architectures with the ones produced by undergraduate students.

18.1 Introduction

A fundamental question of computing is: "What can be automated?" [1]. Is software architecture design inherently a human activity, sensitive to all human weaknesses, or could it be automated to a certain degree? Given functional and quality requirements for a particular system, could it be possible to generate a reasonable software architecture design for the system automatically, thus avoiding human pitfalls (like the Golden Hammer syndrome [2])? Besides being interesting from the viewpoint of understanding the limits of computing and the character of software architecture, we see answers to these questions relevant from a pragmatic viewpoint as well. In particular, if it turns out that systems can successfully design systems, various kinds of software generators can optimize the architecture according to the application requirements, self-sustaining systems [3] can dynamically improve their own architecture in changing environments, and architects can be supported by automated design tools.

A possible approach to automate software architecture design is to mechanize the human process of architecture design into a tool that selects or proposes architectural solutions using similar rules as a human would. A good example of this approach is ArchE, a semi-automated assistant for architecture design [4]: the design knowledge

P. Avgeriou et al. (eds.), *Relating Software Requirements and Architectures*,
DOI 10.1007/978-3-642-21001-3_18, © Springer-Verlag Berlin Heidelberg 2011

is codified as a reasoning framework that is applied to direct the design process. In this approach, the usefulness of the resulting architecture largely depends on the intelligence and codified knowledge of the tool. This is a potential weakness as far as automation is concerned: it is hard to capture sufficient design knowledge in a reasoning framework, which decreases the automation level.

An alternative approach is not to mechanize the process and rules of architecture design, but simply give certain criteria for the "goodness" of an architecture, and let the tool try to come up with an architecture proposal that is as good as possible, using whatever technique. This approach requires no understanding of the design process, but on the other hand it requires a characterization of a "good" architecture. If suitable metrics can be developed for different quality attributes of software architectures, this approach is more light-weight than the former. In particular, this approach is more amenable for an automated design process, as it can be presented essentially as a search problem. We will briefly review some existing work related to this approach in Sect. 18.3.

In this chapter, we will follow the latter approach. More precisely, we will make the following assumptions to simplify the research setup. First, we assume that the architecture synthesizer can rely on a "null architecture" that gives the basic decomposition of the functionalities into components, but pays no attention to the quality requirements. We will later show how the null architecture is derived from use cases. Second, we assume that the architecture is obtained by augmenting the null architecture with applications of general architectural solutions. Such solutions are typically architectural styles and design patterns [5]. Third, we assume that the goodness of an architecture can be inferred by evaluating a representation of the architecture mechanically against the quality requirements of the system. Each application of a general solution enhances certain quality attributes of the system, at the expense of others.

With these assumptions, software architecture design becomes essentially a search problem: find a combination of applications of the general solutions that satisfies the quality requirements in an optimal way. However, given multiple quality attributes and a large number of general solutions, the search space becomes huge for a system with realistic size. This leads us to the more refined research problem discussed in this chapter: to what extent could we use meta-heuristic search methods, like genetic algorithms (GA) [6, 7], to produce a reasonable software architecture automatically for certain functional and quality requirements?

The third assumption above is perhaps the most controversial. Since there is no exact definition of a good software architecture, and different persons would probably in many cases disagree on what is a good architecture, this assumption means that we can only approximate the evaluation of the goodness. Obviously, the success of a search method depends on how well we can capture the intuitive architecture quality in a formula that can be mechanically evaluated.

In this paper, we consider three quality attributes, modifiability, efficiency, and simplicity; these correspond roughly to the ISO9126 quality factors changeability, time behavior and understandability [8], respectively. We base our evaluation of all these factors on existing software metrics [9], but extend them for modifiability and

efficiency by exploiting knowledge about the effect of the solutions on these two quality factors. Optional information given by the designer about certain functionalities is also taken into account. In addition, the designer can give more precise modifiability requirements as change scenarios [10], taken into account in the evaluation of modifiability as well.

Although a number of heuristic search methods could be used here [11], we are particularly interested in GA for two main reasons. First, the structural solutions visible in the living species in nature provide an indisputable evidence of the power of evolution in finding competitive system architectures. Second, crossover can be naturally interpreted for software architecture, as long as certain consistency rules are followed. Crossover can be viewed as a situation where two architects provide alternate designs for a system, and decide to merge their solutions, (hopefully) taking the best parts of both designs.

This chapter proceeds as follows. Background information on genetic algorithms and the proposed evolutionary software architecture generation process are discussed in Sect. 18.2. Existing work related to search-based approaches to software architecture design is briefly reviewed in Sect. 18.3. The GA realization in our approach is discussed in Sect. 18.4, concretized with an example system. An application of the technique for the example system is presented in Sect. 18.5, and an empirical experiment evaluating the quality of the genetically produced software architecture is discussed in Sect. 18.6. Finally, we conclude with some remarks about the implications of the results and future directions of our work.

18.2 Background

18.2.1 Genetic Algorithms

Meta-heuristics [12] are commonly used for combinatorial optimization, where the search space can become especially large. Many practically important problems are NP-hard, making exact algorithms not feasible. Heuristic search algorithms handle an optimization problem as a task of finding a “good enough” solution among all possible solutions to a given problem, while meta-heuristic algorithms are able to solve even the general class of problems behind the certain problem. A search will optimally end in a global optimum in a search space, but at the very least it will give some local optimum, i.e., a solution that is “better” than alternative solutions nearby. A solution given by a heuristic search algorithm can be taken as a starting point for further searches or be taken as the final solution, if its quality is considered high enough.

We have used genetic algorithms, which were invented by John Holland in the 1960s. Holland’s original goal was not to design application specific algorithms, but rather to formally study the ways of evolution and adaptation in nature and develop ways to import them into computer science. Holland [6] presents the genetic

algorithm as an abstraction of biological evolution and gives the theoretical framework for adaptation under the genetic algorithm.

In order to explain genetic algorithms, some biological terminology needs to be clarified. All living organisms consist of *cells*, and every cell contains a set of *chromosomes*, which are strings of DNA and give the basic information of the particular organism. A chromosome can be further divided into *genes*, which in turn are functional blocks of DNA, each gene representing some particular property of the organism. Each gene is located at a particular *locus* of the chromosome. When reproducing, *crossover* occurs and genes are exchanged between the pair of parent chromosomes. The offspring is subject to *mutation* where single bits of DNA are changed. The *fitness* of an organism implies the probability that the organism will live to reproduce and carry on to the next generation [7]. The set of chromosomes at hand at a given time is called a *population*.

During the evolution, the population needs to change to fit better the requirements of the environment. The changing is enabled by mutations and crossover between different chromosomes (i.e., individuals), and, due to natural selection, the fittest survive and are able to participate in creating the next generation.

Genetic algorithms are a way of using the ideas of evolution in computer science to find a good solution from a very large search space, the goal obviously being that the found solution is as good as possible. To operate with a genetic algorithm, one needs an encoding of the solution, i.e., a representation of the solution in a form that can be interpreted as a chromosome, an initial population, mutation and crossover operators, a fitness function (to determine the "goodness" of a solution) and a selection operator for choosing the survivors for the next generation.

In addition, there are also many parameters regarding the GA that need to be defined and greatly affect the outcome. These parameters are the population size, number of generations (often used as the terminating condition) and the mutation and crossover probabilities. Having a large enough population ensures variability within a generation, and enables a wide selection of different solutions at every stage of evolution. However, a larger population always means more fitness evaluations and thus requires more computation time. Similarly, the more generations the algorithm is allowed to run, the higher the chances are that it will be able to reach the global optimum. However, again, letting an algorithm run for, say, 10,000, generations will most probably not be beneficial: if the operations and parameters have been chosen correctly, a reasonably good solution should have been found much earlier.

Mutation and crossover probabilities both affect the speed of evolution. If the probabilities are too high, there is the risk that the application of genetic operations becomes random instead of guided. Vice versa, if the probabilities are too low there is the risk that the population will evolve too slowly, and no real diversity will exist. A theory to be noted with genetic operators is the building block hypothesis, which states that a genetic algorithm combines a set of sub-solutions, or building blocks, to obtain the final solution. The sub-solutions that are kept over the generations usually have an above-average fitness [13]. The crossover operator is especially

sensitive to this hypothesis, as an optimal crossover would thus combine two rather large building blocks in order to produce an offspring.

18.2.2 Overview of Evolutionary Software Architecture Generation

Software architecture can be understood in different ways. The definitions of software architecture usually cover the high-level structure of the system, but in addition to that, often also more process-related aspects like design principles and rationale of design decisions are included [14, 15]. To facilitate our research, we adopt a narrow view of software architecture, considering only the static structural aspect, expressible as a UML (stereotyped) class diagram. In terms of the 4 + 1 views of software systems [16], this corresponds to a (partial) logical view. While a similar approach could be applied to generate other views of software architectures as well, there are some fundamental limitations in using heuristic methods. For example, it is very difficult to produce the rationale for the design decisions proposed by a heuristic method.

A central issue in our approach is the representation of the functional and quality requirements of the system, to be given as input for the genetic synthesis of the architecture. For expressing functional requirements we need to identify and express the primary use cases of the system, and refine them into sequence diagrams depicting the interaction between major components required to accomplish the use cases. This is a manual task, as the major components have to be decided, typically based on domain analysis.

In our approach, a so-called null architecture represents a basic functional decomposition of the system, given as a UML class diagram. No quality requirements are yet taken into account in the null architecture, although it does fulfill the functional requirements. The null architecture can be systematically derived from the use case sequence diagrams: the (classes of the) participants in the sequence diagram become the classes, the operations of a class are the incoming call messages of the participants of that class, and the dependency relationships between the classes are inferred from the call relationships of the participants. This kind of generation of a class diagram can be automated [17], but in the experiments discussed here we have done this manually.

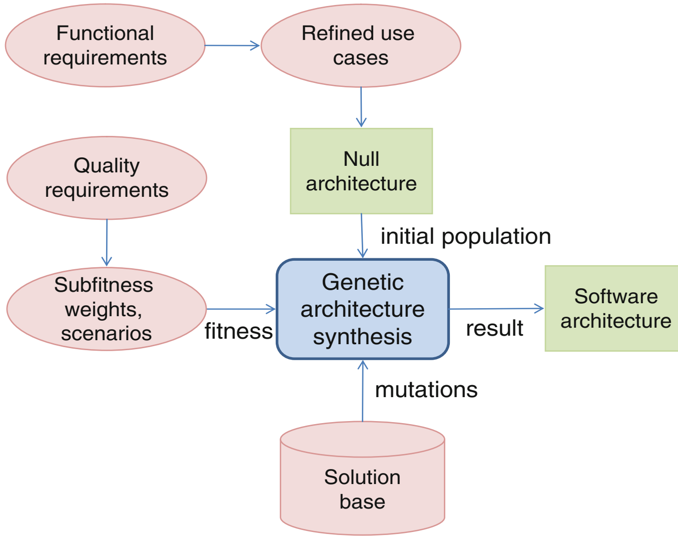
Depending on the quality attributes considered, various kinds of information may need to be associated with the operations of the null architecture. In our study we consider three quality attributes: simplicity, modifiability, and efficiency. Simplicity is an operation-neutral property in the sense that the characteristics of the operations have no effect on the evaluation of simplicity. In contrast, modifiability and efficiency are partially operation-sensitive. For evaluating the modifiability of a system, it is useful to know which operations are more likely to be affected by changes than others. Similarly, for evaluating efficiency it is often useful to know

something about the frequency and resource consumption of the operations. For example, if an operation that is frequently needed is activated via a message dispatcher, there is a performance cost because of the increased message traffic. To allow the evaluation of modifiability and efficiency, the operations can be annotated with this kind of optional information. If this information is insufficient, the method may produce less satisfactory results than with the additional information. However, no actual "hints" on how the GA should proceed in the design process are given. The null architecture gives a skeleton for the system and does not give any finer details regarding the architectures. The information regarding the operation merely helps in evaluating the solutions but influences in no direct way the choices of the GA.

The specific quality requirements of a system are represented in two ways. First, the fitness function used in the GA is basically a weighted sum of the values of individual quality attributes. By changing the weights the user can emphasize or downplay some quality attributes, or remove completely certain quality attributes as requirements. Second, the user can optionally provide more specific quality requirements using so-called scenarios. The scenario concept is inspired by the ATAM architecture evaluation method [10], where scenarios are imaginary situations or sequences of events serving as test cases for the fulfilling of a certain quality requirement. In principle, scenarios could be used for any quality attribute, but their formalization is a major research issue outside the scope of this work. Here we have used only modifiability scenarios, which are fairly easy to formalize. For example, in our case a scenario could be: "With 50% probability operation T needs to be realized in different versions that can be changed dynamically." This is expressed for the GA tool using a simple formal convention covering most usual types of change scenario contents.

Figure 18.1 depicts the overall synthesis process. The functional requirements are expressed as use cases, which are refined into sequence diagrams. This is done manually by exploiting knowledge of the major logical domain entities having functional responsibilities. The null architecture, a class diagram, is derived mechanically from the sequence diagrams. The quality requirements are encoded for the GA as a fitness function, which is used to evaluate the produced architectures. Weights can be given as parameters to emphasize certain quality attributes, and scenarios can be used for more specific quality (modifiability) requirements. When the evolution begins, the null architecture is used by the GA to first create an initial population of architectures and then, after generations of evolution, the final architecture proposal is presented as the best individual of the last generation. New generations are produced by applying a fixed library of standard architectural solutions (styles, patterns, etc.) as mutations, and crossover operations to combine architectures. The probabilities of mutations and crossover can be given as parameters as well. The GA part is discussed in more detail in Sect. 18.4.

The influence of human input is present in defining the use cases which lead to the null architecture and giving the parameters for the GA. The use cases must be defined manually, as they depict the functional requirements of a system: automatically deciding what a system is needed for is not sensible. Giving the parameters for



This figure will be printed in b/w

Fig. 18.1 Evolutionary architecture generation

the GA, in turn, is necessary for the algorithm to operate. It is possible to leave everything for the algorithm, and give each mutation the same probability and each part of the fitness function the same weight. In this case, the GA will not favor any design choice or quality aspect over another. If, however, the human architect has a vision that certain design solutions would be more beneficial for a certain system or feels that one quality aspect is more important than some other, it is possible to take these into account when defining the parameters.

Thus, the human restricts the GA in terms of defining the system functionality and guides the GA in terms of defining parameters. Additionally, the GA is restricted by the solution base. The human can influence the solution base by "removing solutions," that is, by giving them probability 0, and thus making it impossible for the GA to use them. But in any case the GA cannot move beyond the solution base: if a pattern is not defined in the solution base, it cannot be used, and thus the design choices are limited to those that can be achieved as a combination of the specified solutions. Currently the patterns must be added to the solution base by manual coding.

18.3 Related Work

Search-based software engineering applies meta-heuristic search techniques to software engineering issues that can be modeled as optimization problems. A comprehensive survey of applications in search-based software engineering has been made by Harman et al. [18]. Recently, there has been increasing interest in

software design in the field of search-based software engineering. A survey on this subfield has been conducted by Räihä [19]. In the following, we briefly discuss the most prominent studies in the field of search-based software design.

Bowman et al. [20] study the use of a multi-objective genetic algorithm (MOGA) in solving the class responsibility assignment problem. The objective is to optimize the class structure of a system through the placement of methods and attributes within given constraints. So far they do not demonstrate assigning methods and attributes "from scratch" (based on, e.g., use cases), but try to find out whether the presented MOGA can fix the structure if it has been modified.

Simons and Parmee [21, 22] take use cases as the starting point for system specification. Data is assigned to attributes and actions to methods, and a set of uses is defined between the two sets. The notion of class is used to group methods and attributes. This approach starts with pure requirements and leaves all designing to the genetic algorithm. The genetic algorithm works by changing the allocation of attributes and methods.

Our work differs from those of Bowman et al. [20] and Simons and Parmee [21, 22] by operating on a higher level. The aforementioned studies concentrate only on class-level structure, and single methods and attributes. Bowman et al. [20] also do not present a method for straightforward design, but are only at the level where the algorithm can correct a set of errors introduced for testing purposes. Simons and Parmee [21, 22] do start from roughly the same level as we do (requirements derived from use cases), but they consider only the assignment of methods and attributes to classes.

Amoui et al. [23] use the GA approach to improve the reusability of software by applying architecture design patterns to a UML model. The authors' goal is to find the best sequence of transformations, i.e., pattern implementations. Used patterns come from the collection presented by Gamma et al. [24]. From the software design perspective, the transformed design of the best chromosomes are evolved so that abstract packages become more abstract and concrete packages in turn become more concrete. When compared to our work, this approach only uses one quality factor (reusability) instead of several contradicting quality attributes. Further, the starting point in this approach is an existing architecture that is more elaborated than our null architecture.

Seng et al. [25] describe a methodology that computes a subsystem decomposition that can be used as a basis for maintenance tasks by optimizing metrics and heuristics of good subsystem design. GA is used for automatic decomposition. If a desired architecture is given, and there are several violations, this approach attempts to determine another decomposition that complies with the given architecture by moving classes around. Seng et al. [26] have continued their work by searching for a list of refactorings, which deal with the placement of methods and attributes and inheritance hierarchy.

O'Keeffe and Ó Cinnéide [27] have developed a tool for improving a design with respect to a conflicting set of goals. The tool restructures a class hierarchy and moves methods within it in order to minimize method rejection, eliminate code duplication and ensure superclasses are abstract when appropriate. Contrary to most

other approaches, this tool uses simulated annealing. O'Keeffe and Ó Cinnéide [28, 29] have continued their research by constructing a tool for refactoring object-oriented programs to conform more closely to a given design quality model. This tool can be configured to operate using various subsets of its available automated refactorings, various search techniques, and various evaluation functions based on combinations of established metrics.

Seng et al. [25, 26] and O'Keeffe and Ó Cinnéide [27–29] make more substantial design modifications than, e.g., Simons and Parmee [21, 22], and are thus closer to our level of abstraction, but they work clearly from the re-engineering point of view, as a well designed architecture is needed as a starting point. Also, modifications to class hierarchies and structures are still at a lower abstraction level than the design patterns and styles we use, as we need to consider larger parts of the system (or even the whole system). The metrics used by Seng et al. [25, 26] and O'Keeffe and Ó Cinnéide are also simpler, as they directly calculate, e.g., the number of methods per class or the levels of abstraction.

Mancoridis et al. [30] have created the Bunch tool for automatic modularization. Bunch uses hill climbing and GA to aid its clustering algorithms. A hierarchical view of the system organization is created based on the components and relationships that exist in the source code. The system modules and the module-level relationships are represented as a module dependency graph (MDG). The goal of the software modularization process is to automatically partition the components of a system into clusters (subsystems) so that the resultant organization concurrently minimizes inter-connectivity while maximizing intra-connectivity.

Di Penta et al. [31] build on these results and present a software renovation framework (SRF) which covers several aspects of software renovation, such as removing unused objects and code clones, and refactoring existing libraries into smaller ones. Refactoring has been implemented in the SRF using a hybrid approach based on hierarchical clustering, GAs and hill climbing, and it also takes into account the developer's feedback. Most of the SRF activities deal with analyzing dependencies among software artifacts, which can be represented with a dependency graph.

The studies by Mancoridis et al. [30] and Di Penta et al. [31] again differ from ours on the direction of design, as they concentrate on re-engineering, and do not aim to produce an architecture from requirements. Also they operate on different design levels: clustering in the case of Mancoridis et al. [30] is on a higher abstraction level, while, e.g., removing code clones in the case of Di Penta et al.'s [31] study is on a much more detailed level than our work.

In the self-adaptation approach presented by Menascé et al. [32], an existing SOA based system is adapted to a changing environment by inserting fault-tolerance and load balancing patterns into the architecture at run time. The new adapted architecture is found by a hill climbing algorithm. This work is close to ours in the use of architecture-level patterns and heuristic search, but this approach – as other self-adaptation approaches – use specific run-time information as the basis of architectural transformations, whereas we aim at synthesizing the architecture based on requirements.

To summarize, most of the approaches discussed above are different from ours in terms of the level of detail and overall aim: we are especially interested to shape the overall architecture genetically, while the works discussed above consider the problem of improving an existing architecture in terms of fairly fine-grained mechanisms.

18.4 Realizing Genetic Algorithms for Software Architecture Generation

18.4.1 Representing Architecture

The genetic algorithm makes use of two kinds of information regarding each operation appearing in the null architecture. First, the basic input contains the call relationships of the operations taken from the sequence diagrams, as well as other attributes like estimated parameter size, frequency and variability sensitiveness, and the null architecture class it is initially placed in. Second, the information gives the position of the operation with respect to other structures: the interface it implements and the design patterns [24] and styles [33] it is a part of. The latter data is produced by the genetic algorithm.

We will discuss the patterns used in this work in Sect. 18.4.2. The message dispatcher architecture style is encoded by recording the message dispatcher the operation uses and the responsibilities it communicates with through the dispatcher. Other patterns are encoded as instances that contain all relevant information regarding the pattern: operations involved, classes and interfaces involved, and whether additional classes are needed for the pattern (as in the case of Façade, Mediator and Adapter). All this data regarding an operation is encoded as a supergene. An example of a supergene representing one operation is given in Fig. 18.2.

The chromosome handled by the genetic algorithm is gained by collecting the supergenes, i.e., all data regarding all operations, thus representing a whole view of the architecture. The null architecture is automatically encoded into the chromosome format on the basis of the sequence diagrams. An example of a chromosome is presented in Fig. 18.3. A more detailed specification of the architecture representation is given by Räihä et al. [34, 35].

calls	name	type	frequency	parameter size	variation	class	interface	dispatcher	dispatcher communications	component class	pattern

This figure will be printed in b/w

Fig. 18.2 A supergene for operation

sg_1	sg_2	....	sg_{n-1}	sg_n

Fig. 18.3 Chromosome for a system with *n* operations (and *n* supergenes)

The initial population is generated by first encoding the null architecture into the chromosome form and creating the desired number of individuals. A random pattern is then inserted into each individual (in a randomly selected place). In addition, a special individual is left in the population where no pattern is initially inserted; this ensures versatility in the population.

18.4.2 Mutations and Crossover

As discussed above, the actual design is made by adding patterns to the architecture. The patterns have been chosen so that there are very high-level architectural styles (message dispatcher and client-server), medium-level design patterns (Façade and Mediator), and low-level design patterns (Strategy, Adapter and Template Method). The particular patterns were chosen also because they mostly deal with structure and need very little or no information of the semantics of the operations involved. The mutations are implemented in pairs of introducing a specific pattern or removing it. The dispatcher architecture style makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the components can communicate through it.

Preconditions are used to check that a pattern is applicable. If, for example, the "add Strategy" –mutation is chosen for operation o_i, it is checked that o_i is called by some other operation in the same class c and that it is not a part of another pattern already (pattern field is empty). Then, a Strategy pattern instance sp_i is created. It contains information of the new class(es) sc_i where the different version(s) of the operation are placed, and the common interface si_i they implement. It also contains information of all the classes and operations that are dependent on o_i, and thus use the Strategy interface. Then, the value in the class field in the supergene sg_i (representing o_i) would be changed from c to sc_i, the interface field would be given value si_i and the pattern field the value sp_i. Adding other patterns is done similarly. Removing a pattern is done in reverse: the operation placed in a "pattern class" would be returned to its original null architecture class, and the pattern found in the supergene's pattern field would be deleted, as well as any classes and interfaces related to it.

The crossover is implemented as a traditional one-point crossover. That is, given chromosomes ch_1 and ch_2 that are selected for breeding, a crossover point p is first chosen at random, so that $0 < p < n$, if the system has n operations. The supergenes $sg_1 \ldots sg_p$ from chromosome ch_1 and supergenes $sg_{p+1} \ldots sg_n$ from ch_2 will form one child, and supergenes $sg_1 \ldots sg_p$ from chromosome ch_2 and supergenes $sg_{p+1} \ldots sg_n$ from ch_1 another child.

A corrective function is added to ensure that the architectures stay coherent, as patterns may be broken by overlapping mutations. In addition to ensuring that the patterns present in the system stay coherent and "legal," the corrective function also checks that no anomalies are brought to the design, such as interfaces without any users.

The mutation (and crossover) points are selected randomly. However, we have taken advantage of the variability property of operations with the Strategy, Adapter and dispatcher communication mutations. The chances of a gene being subjected to these mutations increase with respect to the variability value of the corresponding operation. This should favor highly variable operations.

The actual mutation probabilities are given as input. Selecting the mutation is made with a "roulette wheel" selection [36], where the size of each slice of the wheel is in proportion to the given probability of the respective mutation. Null mutation and crossover are also included in the wheel. The crossover probability increases linearly in relation to the fitness rank of an individual, which causes the probabilities of mutations to decrease in order to fit the larger crossover slice to the wheel. Also, after crossover, the parents are kept in the population for selection. These actions favor strong individuals to be kept intact through generations. Each individual has a chance of reproducing in each generation: if the first roulette selection lands on a mutation, another selection is performed after the mutation has been administered. If the second selection lands on the crossover slice, the individual may produce offspring. In any other case, the second selection is not taken into account, i.e., the individual is not mutated twice.

18.4.3 Fitness Function

The fitness function needs to produce a numerical value, and is thus composed of software metrics [37, 38]. The metrics introduced by Chidamber and Kemerer [9] have especially been used as a starting point for the fitness function, and have been further developed and grouped to achieve clear "sub-functions" for modifiability and efficiency, both of which are measured with a set of positive and negative metrics. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-server architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. A simplicity metric is added to penalize having many classes and interfaces.

Dividing the fitness function into sub-functions gives the possibility to emphasize certain quality attributes and downplay others by assigning different weights for different sub-functions. These weights are set by the human user in order to guide the GA in case one quality aspect is considered more favorable than some other. Denoting the weight for the respective sub-function sf_i with w_i, the core fitness function $f_c(\mathrm{x})$ for architecture x can be expressed as

$$f_c(\mathrm{x}) = \mathrm{w}_1{}^* sf_1 - \mathrm{w}_2{}^* sf_2 + \mathrm{w}_3{}^* sf_3 - \mathrm{w}_4{}^* sf_4 - \mathrm{w}_5{}^* sf_5$$

Here, sf_1 measures positive modifiability, sf_2 negative modifiability, sf_3 positive efficiency, sf_4 negative efficiency and sf_5 complexity. The sub-fitness functions are defined as follows (|X| denotes the cardinality of X):

sf_1 = |interface implementers| + |calls to interfaces| + |calls to server| + |calls through dispatcher| * $\prod$ (variabilities of operations called through dispatcher) – |unused operations in interfaces| * α,
sf_2 = |direct calls between operations in different classes|,
sf_3 = $\sum$ (|operations dependent of each other within same class| * parameterSize) + $\sum$ (|usedOperations in same class| * parameterSize + |dependingOperations in same class| * parameterSize),
sf_4 = $\sum$ ClassInstabilities [23] + (|dispatcherCalls| + |serverCalls|) * $\sum$ frequencies,
sf_5 = |classes| + |interfaces|.

The multiplier α in sf_1 emphasizes that having unused responsibilities in an interface should be more heavily penalized. In sf_3, "usedOperations in same class" means the set of operations $o_i \ldots o_l$ in class C, which are all used by the same operation o_m from class D. Similarly, "dependingOperations in same class" means the set of operations $o_b \ldots o_h$ in class K, which all use the same operation o_a in class L.

It should be emphasized that all these sub-functions calculate a numerical fitness value for the entire system, and do not reward or penalize any specific patterns (apart from dispatcher connections). This fitness value is the basis of the evaluation, and weights are simply used to guide the algorithm, if needed. Each weight can be set to 1, in which case all sub-fitnesses are considered equally important, and the fitness value is the raw numerical value produced by the fitness calculations. All sub-fitnesses are normalized so that their values are in the same range.

Additionally, scenarios can be used for more detailed fitness calculations. Basically, a scenario describes an interaction between a stakeholder and the system [39]. In our approach we have concentrated only on change scenarios. We have categorized each scenario in three ways: is the system changed or is something added; if changed, does the change concern semantics or implementation of the operation, and whether the modification should be done dynamically or statically. This categorization is the basis for encoding the scenarios. In addition, each encoding of a scenario contains information of the operation it affects, and the probability of the scenario occurrence. Räihä et al. [40] explain the scenario encoding in more detail.

Each scenario type is given a list of preferences according to the general guidelines of what is a preferable way to deal with that particular type of modification. These preferences are general, and do not in any way consider the specific needs or properties of the given system.

When scenarios are encoded, the algorithm processes the list of given scenarios, and compares the solution for each scenario to the list of preferences. Each solution is then awarded points according to how well it supports the scenarios, i.e., how high the partial solutions regarding individual operations are on the preference list.

Formally, the scenario sub-fitness function sf_s can be expressed as

$$sf_s = \sum \text{scenarioProbability} * 100/\text{scenarioPreference}.$$

Adding the scenario sub-fitness function to the core fitness function results in the overall fitness, $f(\mathrm{x}) = f_c(\mathrm{x}) + \mathrm{w_s} * sf_s$

18.5 Application

18.5.1 Creating Input

As an example system, we will use the control system for a computerized home, called ehome. Use cases for this system are assumed to consist of logging in, changing the room temperature, changing the unit of temperature, making coffee, moving drapes, and playing music. In Fig. 18.4, the coffee making use case has been refined into a sequence diagram.

Since we are here focusing on the architecture of the actual control system, we ignore user interface issues and follow a simple convention that the user interface is represented by a single (subsystem) participant that can receive use case requests. Accordingly, in the null architecture the user interface is in this example represented by a single component that has the use cases as operations.

To refine this use case, we observe that we need further components. The main unit for controlling the coffee machine is introduced as CoffeeManager; additionally, there is a separate component for managing water, WaterManager. If a component has a significant state or it manages a significant data entity (like, say, a data base), this is added to the participant box. In this case, CoffeeManager and WaterManager are assumed to have significant state information.

The null architecture in Fig. 18.5 (made by hand in this study) for the ehome system can be mechanically derived from the use case sequence diagrams. The null architecture only contains use relationships, as no more detail is given for the algorithm at this point. The null architecture represents the basic functional decomposition of the system.

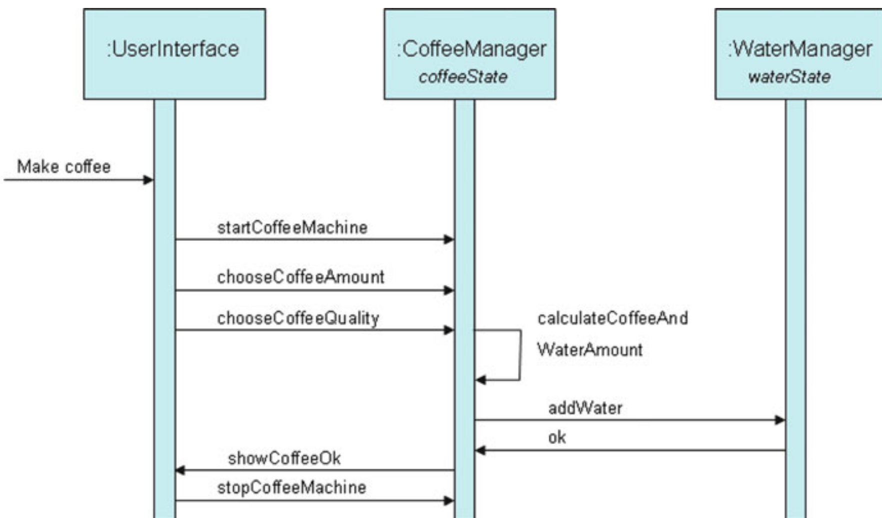


Fig. 18.4 Make coffee use case refined

This figure will be printed in b/w

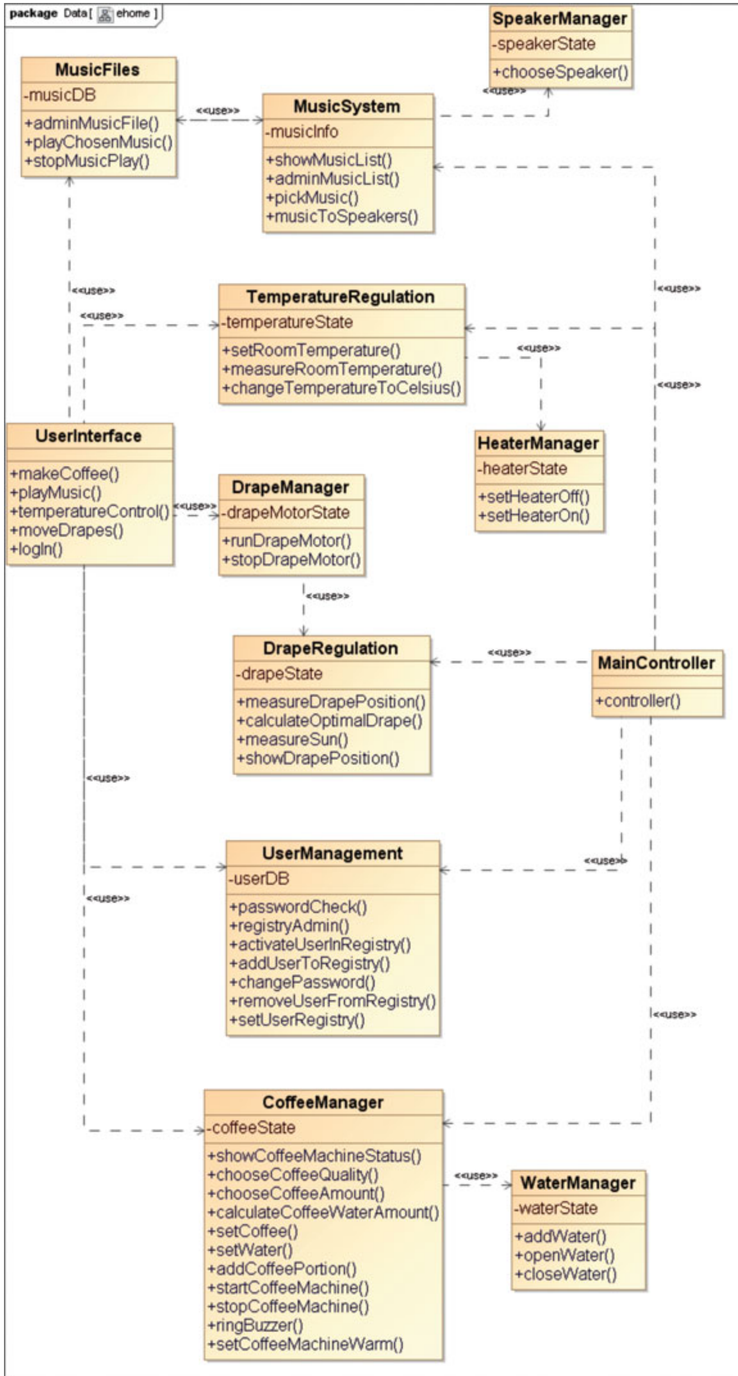


Fig. 18.5 Null architecture for ehome

This figure will be printed in b/w

After the operations are derived from the use cases, some properties of the operations can be estimated to support the genetic synthesis, regarding the amount of data an operation needs, frequency of calls, and sensitiveness for variation. For example, it is likely that the coffee machine status can be shown in several different ways, and thus it is more sensitive to variation than ringing the buzzer when the coffee is done. Measuring the position of drapes requires more information than running the drape motor, and playing music quite likely has a higher frequency than changing the password for the system. Relative values for the chosen properties can similarly be estimated for all operations. This optional information, together with operation call dependencies, is included in the information subjected to encoding.

Finally, different stakeholders' viewpoints are considered regarding how the system might evolve in the future, and modifiability scenarios are formulated accordingly. For example, change scenarios for the ehome system include:

- The user should be able to change the way the music list is showed (90%)
- The developer should be able to change the way water is connected to the coffee machine (50%)
- The developer should be able to add another way of showing the coffee machine status (60%).

A total of 15 scenarios were given for the ehome system.

18.5.2 Experiment

In our experiment, we used a population of 100 and 250 generations. The fitness curve presented is an average of 10 test runs, where the actual y-value is the average of 10 best individuals in a given population. The weights and probabilities for the tests were chosen based on previous experiments [34, 35, 40].

We first set all the weights to 1, i.e., did not favor any quality factor over another. The architecture achieved this way was quite simple. There were fairly well-placed instances of all low-level patterns (Adapter, Template Method and Strategy), and the client-server architecture style was also applied. Strikingly, however, the message dispatcher was not used as the general style, which we would have expected for this type of system. Consequently, we calibrated the weights by emphasizing positive modifiability over other quality attributes. Simultaneously negative efficiency was given a smaller than usual weight, to indicate that possible performance penalty of solutions increasing modifiability is not crucial. The fitness curve for this experiment is given in Fig. 18.6. As can be seen, the fitness curve develops steadily, and most improvement takes place between 1 and 100 generations, which is expected, as the architecture is still simple enough that applying the different mutations is easy.

An example solution with increased modifiability weight is depicted in Fig. 18.7. Now, the dispatcher architecture style is present, and there are also more Strategy patterns than in the solution where all quality factors were equally weighted. This is a natural consequence of the weighting: the dispatcher has a significant positive

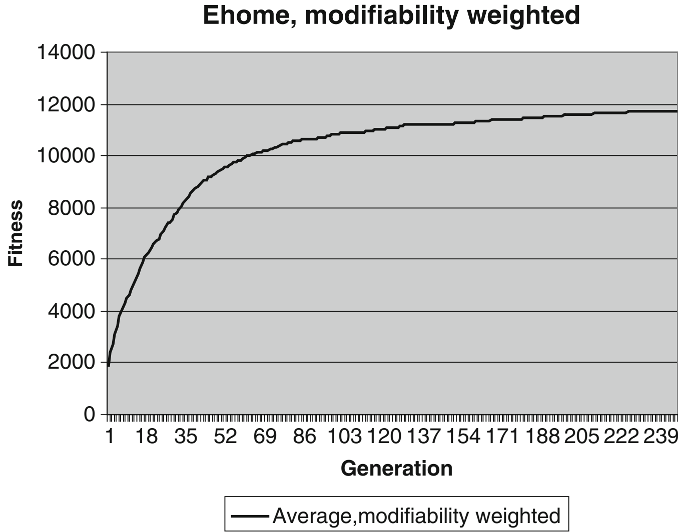


Fig. 18.6 Fitness development, modifiability weighted

effect on modifiability, and since it is not punished too much for inefficiency, it is fairly heavily used as a communication pattern. The same applies to Strategy, although in smaller scale.

18.6 Empirical Study on the Quality of Synthesized Architectures

As shown in the previous section, genetic software architecture synthesis appears to be able to produce reasonable architecture proposals, although obviously they still need some human polishing. However, since the method is not deterministic, it is essential to understand what is the goodness distribution of the proposals, that is, to what extent the architect can rely on the quality of the generated architecture. To study this, we carried out an experiment where we wanted to relate the quality of the generated architectures to the quality of the architectures produced by students. The setup and results of this experiment are discussed in the sequel.

18.6.1 Setup

18.6.1.1 Producing Architectures

First, a group of 38 students from an undergraduate software engineering class was asked to produce an architecture design for the ehome system. Most of the students

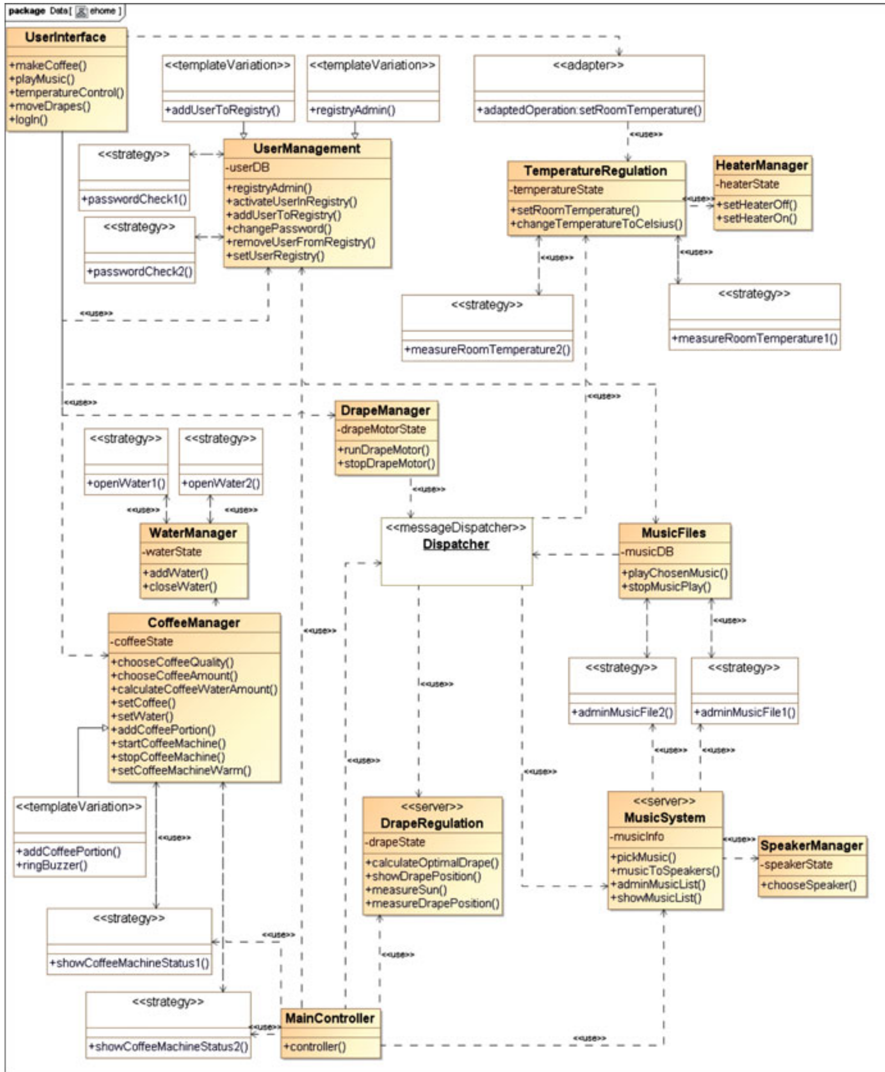


Fig. 18.7 Example architecture for ehome when modifiability is weighted over other quality factors

This figure will be printed in b/w

were third year Software Systems majors from Tampere University of Technology, having participated in a course on software architectures.

The students were given essentially the same information that is used as input for the GA, that is, the null architecture, the scenarios, and information about the expected frequencies of operations. In addition, students were given a brief explanation of the purpose and functionality of the system. They were asked to design the architecture for the system, using only the same architecture styles (message dispatcher and

client-server) and design patterns (Façade, Mediator, Strategy, Adapter, Template Method) that were available for GA. The students were instructed to consider efficiency, modifiability and simplicity in their designs, with an emphasis on modifiability. It took 90 min in the average for the students to produce a design.

In this experiment we wanted to evaluate genetically synthesized solutions against the student solutions in pairs. The synthesized solutions were achieved in 38 runs, out of which ten runs were randomly selected, resulting in ten architecture proposals. Each run took approximately 1 min (i.e., it took 1 min for the synthesizer to produce one solution). The setup for the synthesized architectures was the same as in the example given in Sect. 18.5.

18.6.1.2 Evaluating Architectures

After the students had returned their designs, the assistant teacher for the course (impartial to the GA research) was asked to grade the designs as test answers on a scale of 1–5, five being the highest. The solutions were then categorized according to the points they achieved. From the categories of 1, 3 and 5, one solution for each category was randomly selected. These architectures were presented as grading examples to four software engineering experts. The experts were researchers and teachers at the Department of Software Systems at Tampere University of Technology. They all had a M.Sc. or a Ph.D. degree in Software Systems or in a closely related discipline and several years of expertise from software architectures, gained by research or teaching.

In the actual experiment, the experts were given ten pairs of architectures. One solution in each pair was a student solution, selected randomly from the set of student solutions, and one was a synthesized solution. The solutions were edited in such a way that it was not possible for the experts to know which solution was synthesized. The experts were then asked to give each solution 1, 3 or 5 points. They were given the same information as the students regarding the requirements. The experts were not told how the solutions were achieved, i.e., that they were a combination of student and synthesized solutions. They were merely asked to help in evaluating how good solutions a synthesizer could make.

18.6.2 Results

The scores given by the experts ($e_1 - e_4$) to all the automatically synthesized architectures ($a_1 - a_{10}$) and architectures produced manually by the students ($m_1 - m_{10}$) are shown in Table 18.1. The points in Table 18.1 are organized so that the points given to the synthesized and human-made solutions of the same pair (ai, mi) are put next to each others so the pairwise points are easily seen. The result of each comparison is one of the following

Table 18.1 Points for synthesized solutions and solutions produced by the students

	a_1	m_1	a_2	m_2	a_3	m_3	a_4	m_4	a_5	m_5	a_6	m_6	a_7	m_7	a_8	m_8	a_9	m_9	a_{10}	m_{10}
e_1	3	3	1	3	5	3	1	5	3	1	1	3	3	3	5	3	3	5	3	3
e_2	5	1	3	3	5	1	1	1	3	3	3	5	1	1	3	1	1	1	5	1
e_3	3	3	3	5	3	3	1	3	3	1	3	1	1	3	1	1	3	3	3	1
e_4	3	1	5	3	3	5	3	1	5	1	5	3	3	3	3	1	3	3	5	1

- The synthesized solution is considered better ($a_i > m_i$, denoted later by +)
- The human-made solution is considered better ($m_i > a_i$, denoted later by −), or
- The solutions are considered equal ($a_i = m_i$, denoted latter by 0).

By doing so, we lose some information because one of the solutions is considered simply "better" even in the situation when it receives 5 points while the other receives 1 point. As can be seen in Table 18.1, this happens totally six times. In five of these six cases the synthesized solution is considered clearly better than the human-made solution, and only once vice versa. As our goal is to show that the synthesized solutions are at least as good as the human-made solutions, this lost of information does not bias the results.

The best synthesized solutions appear to be a_3 and a_{10}, with two 3's and two 5's. In solution a_3 the message dispatcher was used, and there were quite few patterns, so the design seemed easily understandable while still being modifiable. However, a_{10} was quite the opposite: the message dispatcher was not used, and there were especially as many as eight instances of the Strategy pattern, when a_3 had only two. There were also several Template Method and Adapter pattern instances. In this case the solution was highly modifiable, but not nearly as good in terms of simplicity. This demonstrates how very different solutions can be highly valued with the same evaluation criteria, when the criteria are conflicting: it is impossible to achieve a solution that is at the same time optimally efficient, modifiable and still understandable.

The worst synthesized solution was considered to be a_4, with three 1's and one 3. This solution used the message dispatcher but also the client-server style was eagerly applied. There were not very many patterns, and the ones that existed were quite poorly applied. Among the human-made solutions, there were three equally scored solutions (m_5, m_8, and m_{10}).

Table 18.2 shows the numbers of the preferences of the experts, with "+" indicating that the synthesized proposal was considered better than the student proposal, "−" indicating the opposite, and "0" indicating a tie. Only one (e_1) of the four experts preferred the human-made solutions slightly more often than synthesized solution, while two experts (e_2 and e_4) preferred the synthesized solutions clearly more often than the human-made solutions. The fourth expert (e_3) preferred both types of solutions equally. There were totally 17 pairs of solutions with better score for the synthesized solution, nine pairs preferring the human-made solution, and 14 ties.

The above crude analysis clearly indicates that in our simple experiment, the synthesized solutions were ranked at least as high as student-made solutions. In order to get more exact information about the preferences and finding confirmation

Table 18.2 Numbers of preferences of the experts

	+	–	0
e_1	3	4	3
e_2	4	1	5
e_3	3	3	4
e_4	7	1	2
Total	17	9	14

even for the hypothesis that the synthesized solutions are significantly better than student-made solutions, it would be possible to use an appropriate statistical test (e.g., counting the Kendall coefficient of agreement). However, we omit such studies due to the small number of both experts and architecture proposals considered. At this stage, it is enough to notice that the synthesized solutions are competitive with those produced by third year software engineering students.

18.6.3 Threats and Limitations

We acknowledge that there are several threats and limitations in the presented experiment. Firstly, as the solutions for evaluations were selected randomly out of all the 38 student (and synthesized) solutions, it is theoretically possible that the solutions selected for the experiment do not give a true representation of the entire solution group. However, we argue that as all experts were able to find solutions they judged worth of 5 points as well as solutions only worth 1 point, and the majority of solutions were given 3 points, it is unlikely that the solutions subjected to evaluation would be so biased it would substantially affect the outcome of the experiment.

Secondly, the pairing of solutions could be questioned. A more diverse evaluation could have been if the experts were given the solutions in different pairs (e.g., for expert e_1 the solution a_1 would have been paired with m_5 instead of m_1). One might also ask if the outcome would be different with different pairing. We argue that as the overall points are better for the synthesized solutions, different pairing would not significantly change the outcome. Also, the experts were not actually told to evaluate the solutions as pairs – the pairing was simply done in order to ease the evaluation and analysis processes.

Thirdly, the actual evaluations made by the experts should be considered. Naturally, having more experts would have strengthened the results. However, the evaluations were quite uniform. There were very few cases where three experts considered the synthesized solution better or equal to the student solution (or the student solution better or equal to the synthesized one) and the fourth evaluation was completely contradicting. In fact, there were only three cases where such contradiction occurred (pairs 2, 3 and 4), and the contradicting expert was always the same (e_4). Thus we argue that the consensus between experts is sufficiently good, and increasing the number of evaluations would not substantially alter the outcome of the experiment in its current form.

Finally, the task setup was limited in the sense that architecture design was restricted to a given selection of patterns. Giving such a selection to the students may both improve the designs (as the students know that these patterns are potentially applicable) and worsen the designs (due to overuse of the patterns). Unfortunately, this limitation is due to the genetic synthesizer in its current stage, and could not be avoided.

18.7 Conclusions

We have presented a method for using genetic algorithms for producing software architectures, given a certain representation of functional and quality requirements. We have focused on three quality attributes: modifiability, efficiency and simplicity. The approach is evaluated with an empirical study, where the produced architectures were given for evaluation to experts alongside with student solutions for the same design problem.

The empirical study suggests that, with the assumptions given in Sect. 18.1, it is possible to synthesize software architectures that are roughly at the level of an undergraduate student. In addition to the automation aspect, major strengths of the presented approach are the versatility and options for expansion. Theoretically, an unlimited amount of patterns can be used in the solution library, while a human designer typically considers only a fairly limited set of standard solutions. The genetic synthesis is also not tied to prejudices, and is able to produce fresh, unbiased solutions that a human architect might not even think of. On the other hand, the current research setup and experiments are still quite limited. Obviously, the relatively simple architecture design task given in the experiment is still far from real-life software architecture design, with all its complications.

The main challenge in this approach is the specification of the fitness function. As it turned out in the experiment, even experts can disagree on what is a good architecture. Obviously, the fitness function can only approximate the idea of architectural quality. Also, tuning the parameters (fitness weights and mutation probabilities) is nontrivial and may require calibration for a particular type of a system. To alleviate the problem of tuning the weights of different quality attributes, we are currently exploring the use of Pareto optimality [41] to produce multiple architecture proposals with different emphasis of the quality attributes, instead of a single one.

In the future we will focus on potential applications of genetic software architecture synthesis. A particularly attractive application field of this technology is self-adapting systems (e.g., Cheng et al. [42]), where systems are really expected to “redesign” themselves without human interaction. Self-adaptation is required particularly in systems that are hard to maintain in a traditional way, like constantly running embedded systems or highly distributed web systems. We see the genetic technique proposed in this paper as a promising approach to give systems the ability to reconsider their architectural solutions based on some changes in their requirements or environment.

Acknowledgments We wish to thank the students and experts for participating in the experiment. The anonymous reviewers have significantly helped to improve the paper. This work has been funded by the Academy of Finland (project Darwin).

References

1. Denning P, Comer DE, Gries D, Mulder MC, Tucker A, Turner AJ, Young PR (1989) Computing as a discipline, Commun. ACM 32(1):9–23
2. Brown WJ, Malveau C, McCormick HW, Mowbray TJ (1998) Antipatterns – refactoring software, architectures, and projects in crisis. Wiley
3. S3 (2008) Proceedings of the 2008 workshop on self-sustaining systems. S3’2008, Potsdam, Germany, 15–16 May 2008. Lecture notes in computer science, vol 5146. Springer-Verlag, Heidelberg
4. Diaz-Pace A, Kim H, Bass L, Bianco P, Bachmann F (2008) Integrating quality-attribute reasoning frameworks in the ArchE design assistant. In: Becker S, Plasil F, Reussner R (eds) Proceedings of the 4th international conference on quality of software-architectures: models and architectures. Lecture notes in computer science, vol 5281. Springer, Karlsruhe, Germany, p 171
5. Buschmann F, Meunier R, Rohnert H, Sommerland P, Stal M (1996) A system of patterns – pattern-oriented software architecture. John Wiley & Sons, West Sussex, England
6. Holland JH (1975) Adaption in natural and artificial systems. MIT Press, Ann Arbor, Michigan, USA
7. Mitchell M (1996) An introduction to genetic algorithms. MIT Press, Cambridge
8. ISO (2001) Software engineering – product quality – part I: quality model. ISO/TEC 9126–1:2001
9. Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE T Software Eng 20(6):476–492
10. Clements P, Kazman R, Klein M (2002) Evaluating software architectures. Addison-Wesley, Reading
11. Clarke J, Dolado JJ, Harman M, Hierons R, Jones MB, Lumkin M, Mitchell B, Mancoridis S, Rees K, Roper M, Shepperd M (2003) Reformulating software engineering as a search problem. IEE Proc – Softw 150(3):161–175
12. Glover FW, Kochenberger GA (eds) (2003) Handbook of metaheuristics, vol 57, International series in operations research & management science. Springer, Heidelberg
13. Salomon R (1998) Short notes on the schema theorem and the building block hypothesis in genetic algorithms. In: Porto VW, Saravanan N, Waagen D, Eiben AE (eds) Evolutionary programming VII, 7th international conference, EP98, California, USA. Lecture notes in computer science, vol 1447. Springer, Berlin, p 113
14. Babar MA, Dingsoyr T, Lago P, van Vliet H (eds) (2009) Software architecture knowledge management – theory and practice establishing and managing knowledge sharing networks. Springer, Heidelberg
15. ISO (2010) Systems and software engineering – architecture description. ISO/IEC CD1 42010: 1–51
16. Kruchten P (1995) Architectural blueprints – the “4 + 1” view model of software architecture. IEEE Softw 12(6):42–50
17. Selonen P, Koskimies K, Systä T (2001) Generating structured implementation schemes from UML sequence diagrams. In: QiaYun L, Riehle R, Pour G, Meyer B (eds) Proceedings of TOOLS 2001. IEEE CS Press, California, USA, p 317
18. Harman M, Mansouri SA, Zhang Y (2009) Search based software engineering: a comprehensive review of trends, techniques and applications. Technical report TR-09-03, Kings College, London

19. Räihä O (2010) A survey on search-based software design. Comput Sci Rev 4(4):203–249
20. Bowman M, Brian, LC, Labiche Y (2007) Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. Technical report SCE-07-02, Carleton University
21. Simons CL, Parmee IC (2007a) Single and multi-objective genetic operators in object-oriented conceptual software design. In: Proceedings of the genetic and evolutionary computation conference (GECCO'07). ACM Press, London, UK, p 1957
22. Simons CL, Parmee IC (2007) A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design. Eng Optim 39(5):631–648
23. Amoui M, Mirarab S, Ansari S, Lucas C (2006) A GA approach to design evolution using design pattern transformation. Int J Inform Technol Intell Comput 1:235–245
24. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns, elements of reusable object-oriented software. Addison-Wesley, Reading
25. Seng O, Bauyer M, Biehl M, Pache G (2005) Search-based improvement of subsystem decomposition. In: Proceedings of the genetic and evolutionary computation conference (GECCO'05). ACM Press, Mannheim, Germany, p 1045
26. Seng O, Stammel J, Burkhart D (2006) Search-based determination of refactorings for improving the class structure of object-oriented systems. In: Proceedings of the genetic and evolutionary computation conference (GECCO'06). ACM Press, Washington, USA, p 1909
27. O'Keeffe M, Ó Cinnéide M (2004) Towards automated design improvements through combinatorial optimization. In: Workshop on directions in software engineering environments (WoDiSEE2004), Workshop at ICSE'04, 26th international conference on software engineering. Edinburgh, Scotland, p 75
28. O'Keeffe M, Ó Cinnéide M (2006) Search-based software maintenance. In: Proceedings of conference on software maintenance and reengineering. IEEE CS Press, Bari, Italy, p 249
29. O'Keeffe M, Ó Cinnéide M (2008) Search-based refactoring for software maintenance. J Syst Software 81(4):502–516
30. Mancoridis S, Mitchell BS, Rorres C, Chen YF, Gansner ER (1998) Using automatic clustering to produce high-level system organizations of source code. In: Proceedings of the international workshop on program comprehension (IWPC'98). Silver Spring, p 45
31. Di Penta M, Neteler M, Antoniol G, Merlo E (2005) A language-independent software renovation framework. J Syst Software 77:225–240
32. Menascé DA, Sousa JP, Malek S, Gomaa H (2010) QoS architectural patterns for self-architecting software systems. In: Proceedings of the 7th international conference on autonomic computing and communications. ACM Press, Washington DC, USA, p 195
33. Shaw M, Garlan D (1996) Software architecture – perspectives on an emerging discipline. Prentice Hall, Englewood Cliffs
34. Räihä O, Koskimies K, Mäkinen E (2008) Genetic synthesis of software architecture. In: Li X et al. (eds) Proceedings of the 7th international conference on simulated evolution and learning (SEAL'08). Lecture notes in computer science, vol 5361. Springer, Melbourne, Australia, p 565
35. Räihä O, Koskimies K, Mäkinen E, Systä T (2008) Pattern-based genetic model refinements in MDA. Nordic J Comput 14(4):338–355
36. Michalewicz Z (1992) Genetic algorithms + data structures = evolutionary programs. Springer, New York
37. Losavio F, Chirinos L, Matteo A, Lévy N, Ramdane-Cherif A (2004) ISO quality standards measuring architectures. J Syst Software 72:209–223
38. Mens T, Demeyer S (2001) Future trends in evolution metrics. In: Proceedings of international workshop on principles of software evolution. ACM Press, Vienna, Austria, p83
39. Bass L, Clements P, Kazman R (1998) Software architecture in practice. Addison-Wesley, Boston

40. Räihä O, Koskimies K, Mäkinen E (2009) Scenario-based genetic synthesis of software architecture. In: Boness K, Fernandes JM, Hall JG, Machado RJ, Oberhauser R (eds) Proceedings of the 4th international conference on software engineering advances (ICSEA'09). IEEE, Porto, Portugal, p 437
41. Deb K (1999) Evolutionary algorithms for multicriterion optimization in engineering design. In: Miettinen K, Mäkelä MM, Neittaanmäki P, Periaux J (eds) Proceedings of the evolutionary algorithms in engineering and Computer Science (EUROGEN'99). University of Jyväskylä, Finland, p 135
42. Cheng B, de Lemos R, Giese H, Inverardi P, Magee J (2009) Software engineering for self-adaptive systems: a research roadmap. In: Cheng BHC, Lemos R de, Giese H, Inverardi P, Magee J (eds) Software engineering for self-adaptive systems. Lecture notes in computer science, vol 5525. Springer, Amsterdam, p 1

Paper IX

Outi Räihä, Kai Koskimies and Erkki Mäkinen, Generating Software Architecture Spectrum with Multi-Objective Genetic Algorithms, In *Proc. of the Third World Congress in Nature and Biologically Inspired Computation (NaBIC'11)*, Salamancá, Spain. October 2011, IEEE Press, to appear.

© 2011 IEEE. Reprinted, with permission

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University'of Tampere's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this material, you agree to all provisions of the copyright laws protecting it.

Generating Software Architecture Spectrum with Multi-Objective Genetic Algorithms

Outi Räihä, Kai Koskimies
Department of Software Systems
Tampere University of Technology
Tampere, Finland
{outi.raiha, kai.koskimies}@tut.fi

Erkki Mäkinen
School of Information Sciences
University of Tampere
Tampere, Finland
erkki.makinen@uta.fi

***Abstract*—A possible approach to partly automated software architecture design is the application of heuristic search methods like genetic algorithms. However, traditional genetic algorithms use a single fitness function with weighted terms for different quality attributes. This is inadequate for software architecture design that has to satisfy multiple incomparable quality requirements simultaneously. To overcome this problem, the use of Pareto optimality is proposed. This technique is studied in the presence of two central quality attributes of software architectures, modifiability and efficiency. The technique produces a spectrum of architecture proposals, ranging from highly modifiable (and less efficient) to highly efficient (and less modifiable). The technique has been implemented and evaluated using an example system. The results demonstrate that Pareto optimality has potential for producing a sensible set of architectures in the efficiency-modifiability space.**

***Keywords*— Pareto optimality; multi-objective genetic algorithm; software design; search-based software engineering; software architecture**

I. INTRODUCTION

Software architecture design has been traditionally regarded as an art rather than as a systematic process: the software architect applies his or her past experiences of working solutions and general architectural knowledge to satisfy the requirements of the target system, but very little is known about the actual thought process that ends up with a particular architecture. Despite recent efforts to systematize software architecture design [9, 13], the process of software architecture design is still insufficiently understood. In particular, it would be important to explore the possibilities and limits of automated software architecture synthesis.

To simplify the research setup, we adopt here the viewpoint that software architecture design can be divided into two successive phases: the functional decomposition of the system into major components, and the application of standard solutions (like architectural styles [19] and design patterns [5, 10]) to satisfy the various quality requirements of the system. The former phase is assumed to be carried out based on domain knowledge and functional analysis (e.g., CRC [3]), resulting in what we call the null architecture. The null architecture satisfies the functional requirements of the system, but so far pays no attention to the quality requirements. Our main focus here is on the second phase: to what extent could it be possible to automate the insertion of various pattern-like solutions to the null architecture, so as to satisfy the quality requirements of the system as completely as possible?

Stating the research problem in this way, an attractive approach to tackle this question is to apply meta-heuristic search methods [11] to find a combination of standard solutions that satisfies the quality requirements in a near-optimal way. Räihä et al. [16, 17] have previously studied applying genetic algorithms (GA) with a simple weighted fitness function for this purpose. Their initial tests [18] suggest that it is possible to achieve roughly the level of a third year software engineering student in terms of the overall quality of the genetically synthesized architecture.

However, software architecture has to satisfy conflicting and competing quality requirements imposed on the system. For example, a software system should often be efficient and easy to modify at the same time. Unfortunately, increasing modifiability usually degrades efficiency and vice versa. In the context of genetic algorithms, this problem is reflected in the formulation of the fitness function.

Rather than producing a single architecture proposal using a weighted fitness function, a more appealing approach is to produce a spectrum of proposals ranging from very modifiable but inefficient architecture to very efficient but inflexible architecture. Then the architect can browse the proposals, compare architectures with different emphasis of modifiability and efficiency, and select a candidate at an appropriate point in the solution spectrum as a jump start of architecture design.

This kind of multi-objective optimization can be achieved using Pareto optimality [8]: instead of using a single fitness value the solutions are ranked multi-dimensionally. Roughly speaking, a solution (here architecture) is Pareto optimal if there is no other solution that is "properly better", i.e., properly better in one property and at least equally good with respect to all other properties. Typically there is not a single Pareto optimal solution but a set of such solutions, the so-called Pareto front. This front will be the spectrum of architecture proposals we are looking for. In this paper we extend our previous studies in genetic

synthesis of software architecture by employing Pareto optimality. Although Pareto optimality can be applied for an arbitrary number of dimensions, we will focus here on two central architectural quality dimensions, modifiability and efficiency.

Validating the results of this kind of research is challenging. Even though the Pareto approach does produce a spectrum of best solutions in the present population in the modifiability-efficiency scale in terms of the fitness function, does it produce such a spectrum in terms of "real" architectural quality?

In software industry, the quality of a software architecture is typically evaluated using scenario-based methods, most notably ATAM (Architecture Trade-off Analysis Method [7]). The basic idea of ATAM is to ask the stakeholders to come up with concrete situations (so-called scenarios) which test a given quality attribute. The architecture is then analyzed against the scenarios. Roughly, the better the architecture supports the scenarios given for a particular quality attribute, the better quality the architecture has with respect to this quality attribute.

To evaluate the results of our work, we have genetically synthesized a spectrum of architecture proposals for a representative example system using the Pareto approach. Then we have imitated an ATAM evaluation for the resulting Pareto front architectures. If this kind of evaluation yields a similar distribution in the modifiability-efficiency space as the original fitness-based distribution, we can conclude that the synthesized Pareto front actually matches with human understanding of the quality of those architectures.

II. RELATED WORK

Recently, approaches dealing with high level structures, such as design patterns and architectures have gained interest in search-based software engineering [11, 15]. We will here concentrate on the ones that are the most related to upstream design, rather than the copious studies closely related to software refactoring and maintenance.

Bowman et al. [4] study the use of a multi-objective genetic algorithm (MOGA) in solving the class responsibility assignment problem. The objective is to optimize the class structure of a system through the placement of methods and attributes within given constraints. So far, they do not demonstrate an upstream solution to the problem, but try to find out whether the presented MOGA can fix the structure if it has been modified.

Simons and Parmee [20] take use cases as the starting point for system specification. Data is assigned to attributes and actions to methods, and a set of uses is defined between the two sets. The notion of class is used to group methods and attributes. Design solutions are encoded directly into an object-oriented programming language.

Amoui et al. [2] use the GA approach to improve the reusability of software by applying architecture design patterns to a UML model. The authors' goal is to find the best sequence of transformations, i.e., pattern implementations. When compared to our work, this approach only uses one quality factor (reusability), and the starting point in this approach is an existing architecture that is more elaborated than our null architecture.

Räihä et al. [16, 17, 18] have taken a pattern-oriented approach that aims at complete software architecture synthesis based on refined use cases, applying GA for finding an optimal combination of patterns. Patterns are used as mutations, and various quality metrics are used for deriving a fitness function.

An even higher level approach is studied by Aleti et al. [1] using AADL (Architecture Analysis and Description Language) models as a basis, and attempting to optimize the architecture with respect to data transfer reliability and communication overhead. They use GA with Pareto optimality in their ArcheOptrix tool, but they concentrate on the optimal deployment of software components to a given hardware platform rather than on the actual software architecture.

Pareto optimality has been applied surprisingly seldom in search-based software engineering taking into account the fact that most problems in software engineering contain conflicting goals. Harman et al.'s survey [11] mentions less than a dozen references that use Pareto optimality. Moreover, most of them are on areas of software engineering which seem remote from the topics of the present paper. A study worth mentioning and using Pareto optimality is that by Harman and Tratt [12] on refactoring, applying a variant of the hill-climbing algorithm to create the Pareto front.

III. GENETIC SYNTHESIS OF SOFTWARE ARCHITECTURE

In this section we describe our approach to synthesize software architecture using genetic algorithms. We assume that the reader is familiar with the basics of genetic algorithms, as given, e.g., by Michalewicz [14]. In what follows, we explain the construction of the initial functional decomposition (null architecture), an encoding of architectures, an initial population, and mutation and crossover operators. The overall synthesis process is depicted in Figure 1. Functional requirements are expressed as use cases which are automatically transformed into a null architecture, and quality requirements are encoded into various parameters affecting the fitness function. Mutation patterns are provided by a subsolution repository.

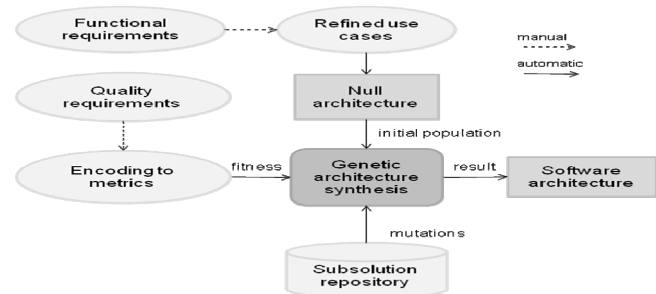


Figure 1. Synthesis process

The fitness function and the selection operator for each new generation will be discussed in the next section in the context of Pareto optimality.

A. *Requirements*

For expressing functional requirements we identify and express the primary use cases of the system, and refine them into sequence diagrams depicting the interaction between major components required to accomplish the use cases. This is a manual task, as the major components have to be decided, typically based on domain analysis.

In our approach, a so-called null architecture represents a basic functional decomposition of the system, given as a UML class diagram. The null architecture can be mechanically derived from the use case sequence diagrams: the (classes of the) participants in the sequence diagram become the classes, the operations of a class are the incoming call messages of the participants of that class, and the dependency relationships between the classes are inferred from the call relationships of the participants. Additionally, if a component has a significant state or it manages a significant data entity (e.g., a data base), this data entity will become an attribute of the class.

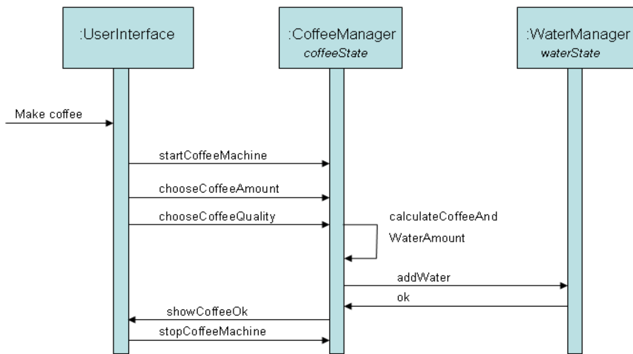


Figure. 2. Make coffee use case refined

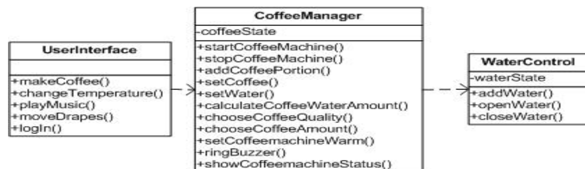


Figure 3. A fragment of the null architecture for ehome

Our example system, the control system of an electronic home (called hereafter ehome), is a typical embedded system. The ehome system controls various devices, providing an interface to allow the user to manage the home. Five distinct subsystems can be identified: user registry, coffee machine, temperature control, drape control and music system. The functional requirements for ehome lead to 56 operations and 90 dependencies between the operations. The resulting null architecture for ehome contains 12 classes. Use cases for ehome can be, for example, logging in, adjusting the room temperature, making coffee, moving drapes, and playing music. In Figure 2, the coffee making use case has been refined into a sequence diagram. A fragment of the null architecture for ehome is given in Figure 3, representing the same part of the system as Figure 2.

In order to evaluate the system, some attributes of the operations are also given, such as sensitiveness to variation, parameter size and frequency of use. The precise values for these attributes of course cannot be known, but should be estimated in order to calculate the modifiability and efficiency values for the system. The given values for the attributes are relative, rather than absolute. Such relative values can straightforwardly be approximated by comparing different operations, e.g., playing music is most likely used much more frequently than changing password.

B. *Genetic Representation*

When the architectural data is encoded into a chromosome form, two kinds of data are given regarding each operation. Firstly, the basic information contains the operations depending on it, its name, type, frequency of use, parameter size, and sensitiveness to variation. Secondly, there is the information regarding the operation's place in the architecture: the class(es) it belongs to, the interface it implements, the message dispatcher it uses (see Subsection III.C), the operations that call it through the message dispatcher, the design patterns it is a part of, and the class it is assigned to in the null architecture. The message dispatcher is given a separate field as opposed to other patterns for efficiency reasons. All data regarding an operation is encoded as a supergene with a separate field for each data particle [2].

The chromosome handled by the genetic algorithm is gained by collecting the supergenes, i.e., all data regarding all operations. The initial population is made by first creating the desired number of individuals with the null architecture. A random pattern is then inserted into each individual, as a population should not consist entirely of clones. In addition, a special individual with no initial patterns is left in the population. The encoding of requirements to supergenes and chromosomes is discussed in more detail by Räihä et al. [18].

C. *Mutation and Crossover Operations*

The actual architectural design means here the application of various standard architectural solutions called collectively patterns: the result of genetic architecture synthesis is the null architecture augmented with patterns. The patterns have been chosen here to represent solutions on different levels: high-level architectural styles [19] (message dispatcher and client-server), medium-level design patterns [10] (Façade and Mediator), and low-level design patterns [10] (Strategy, Adapter and Template Method). The mutations are implemented in pairs of introducing a specific pattern or removing it. The message dispatcher architecture style makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the operations can communicate through it. Preconditions are applied when introducing mutations in order to avoid conflict with existing architecture.

The crossover operation is implemented as a traditional one-point crossover with a corrective function. This function ensures that the architecture stays coherent, as patterns might otherwise be broken by overlapping mutations.

In addition to ensuring that the patterns present in the system stay coherent and "legal", the corrective function also checks that the design conforms to certain architectural laws that we have defined. These laws demand uniform calls between two classes (e.g., through an interface or a dispatcher), and state some basic rules regarding architectures (e.g., an operation can be accessed through at most one interface). The purpose of these laws is to ensure that no anomalies are brought to the design. We have chosen to use preconditions and a corrective function as opposed to handling malformed solutions in the fitness function to ensure that all solutions are always valid. Since the patterns affect only the quality attributes of the system but not its functional properties, individual architectures are also always valid with respect to functional requirements.

The actual mutation probabilities can be tuned as desired. Selecting the mutation is made with a "roulette wheel" selection [14], where the size of each slice of the wheel is in proportion to the given probability of the respective mutation. Null mutation and crossover are also included in the wheel. Each individual has a chance of reproducing in each generation: if the first roulette selection lands on a mutation, another selection is performed after the mutation has been administered. If the second selection lands on the crossover slice, the individual may produce offspring. In any other case, the second selection is not taken into account, i.e., the individual is not mutated twice.

IV. PARETO OPTIMAL FITNESS AND SELECTION

A. *Pareto Optimality*

Suppose that in a given design task the solutions are measured according to p properties and that F is the set of feasible solutions. For notational convenience, we suppose that all properties are maximized, i.e., the bigger the value, the better is the solution. A solution x can then be described by a vector $x = [f_1(x), f_2(x), \ldots, f_p(x)]$, where $f_i(x)$ is the value of *i*th property in x. In a design task with conflicting goals it is unlikely to find a solution in F which would be optimal with respect to all the properties measured. In such a situation, Pareto optimality gives us a way to compare the solutions [8].

We say that a solution $x^* \in F$ is Pareto optimal if for each $x \in F$, we have either $f_i(x) = f_i(x^*)$, for all $i = 1, \ldots, p$, or there is at least one property i such that $f_i(x) < f_i(x^*)$. That is, x^* is Pareto optimal if there exists no feasible solution x that increases some criterion without causing a simultaneous decrease in at least one other criterion. Typically, there is not a single solution that is Pareto optimal, but a set of Pareto optimal solutions.

Given a set of feasible solutions, its Pareto optimal solutions are said to form a Pareto front. The Pareto front is particularly useful in any design task: by restricting attention to the set of solutions that are Pareto optimal, a designer can make trade-offs within this set, rather than considering the full range of every parameter. (Further material concerning Pareto optimality and Pareto fronts can be found, e.g., from [8].)

In our context, the Pareto front consists of the architectures that are Pareto optimal in the populations created by GA. The algorithm iteratively makes use of the Pareto fronts found in the generations created by the previous step of the algorithm. The final result consists of the Pareto front in the last generation created. Hence, the final result is a set of architectures, a spectrum of architecture proposals the designer can browse and choose the desired emphasis of the different properties (or quality attributes).

B. *Fitness Function and Selection*

The fitness function used here is based on widely used software metrics [6]. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-server architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. The fitness function f(x) for chromosome x can be expressed as a vector of sub-functions

$$f(x) = [sf_1(x), sf_2(x), sf_3(x), sf_4(x)].$$

Here, sf_1 measures positive modifiability, which rewards calls between operations that are handled via interfaces, message dispatcher or server. The sensitiveness to variation of the operations is used to enhance the reward. Negative modifiability is calculated in sf_2 by penalizing direct calls between operations, and is given a coefficient -1 to indicate its negative effect. The sensitiveness to variation is also used here to enhance punishment for bad design choices regarding exceptionally variable operations. The total modifiability value is the sum of positive modifiability and negative modifiability and should be maximized.

As for efficiency, sf_3 measures positive efficiency by rewarding structures which lead to minimal amount of calls between different classes and maximum amount of calls between operations within the same class. The operation's required amount of data is also considered and used to increase the reward. Negative efficiency (sf_4), in turn, counts the relation of calls between classes and within classes, and the amount of calls to the message dispatcher and through servers, which are especially penalized by taking into account the frequency of calls to the operations involved. Negative efficiency, like negative modifiability, is given a coefficient -1 to indicate its negative effect. The total efficiency value is the sum of positive efficiency and negative efficiency, and should be maximized.

Both qualities are also given a normalizing weight so that the value ranges are the same. This also enables comparing the multi-objective results with a single-objective approach, where the fitness function is acquired by simply summing the different sub-functions.

Selecting the individuals for each generation is made as follows: the actual Pareto front pf_1 of the population in *i*:th generation, p_i, is first collected, and stored for the population in the next generation p_{i+1}. However, as the front usually contains less than ten individuals and the population size typically more than 100, just one front is not enough to make a sufficient population. Thus, the Pareto front of the remaining individuals in p_i, i.e., the Pareto front pf_2 of the set $p_i \setminus pf_1$ is selected and moved to p_{i+1}. This process is repeated

until p_{i+1} has at least the required minimum of individuals. This method is similar to the selection methods typically used in the context of Pareto optimality (see. e.g., [8]).

V. EXPERIMENTS

We used the ehome sample system (presented in Section III) to test our approach. In our experiments we used a population size of 100, 250 generations and did 20 test runs. As stated in Section IV, the weights for the different fitness aspects (modifiability and efficiency) were set so that the final values were of the same scale. We will first present the data (Pareto fronts), and then discuss the actual architectures.

The collected Pareto fronts of all 20 runs after 50 generations are presented in Figure 4, and the respective fronts after all 250 generations are given in Figure 5. The figures portray scatter plots, where the total values (explained in Subsection IV.B) for modifiability and efficiency are used, and each series (front) has a distinctive marker.

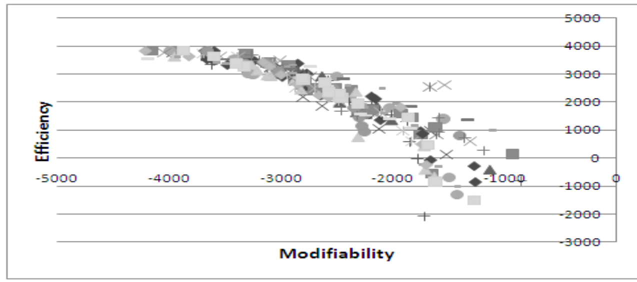


Figure 4. Pareto fronts of 20 runs after 50 generations

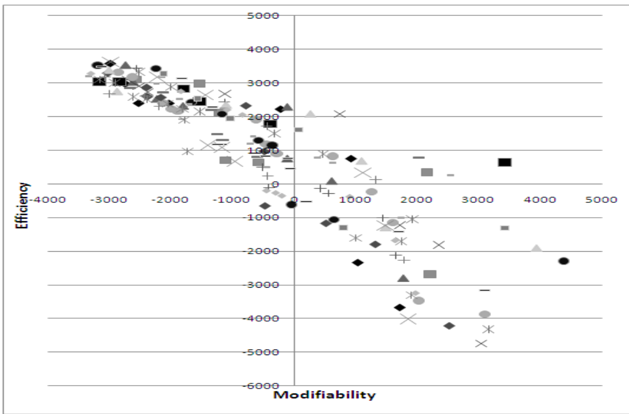


Figure 5. Pareto fronts of 20 runs after 250 generations

In the beginning of evolution (Figure 4), the fronts are quite uniformly located on the upper left-hand side, where the solutions are highly efficient but poor in terms of modifiability. This is expected, as the architecture is most efficient at null architecture stage (generation 0), as the patterns tend to decrease efficiency. When the evolution has ended (Figure 5), the fronts have moved significantly in relation to the x-axis (modifiability). However, efficient solutions are still clustered, which is natural (as discussed above: most efficient solutions are those with minimal amount of mutations applied).

On the right-hand side of Figure 5, where the more modifiable solutions lie, fronts are much more sparse. This is also natural, as modifiability, in turn, can be reached in different ways; every pattern and dispatcher or server connection increases modifiability in its own way, and reduces efficiency accordingly. The amount of different combinations of patterns and dispatcher connections is immensely large, and thus it would be unnatural to have such a clustered Pareto front on the "modifiability side" as can be seen on the "efficiency side". However, despite the scattering of solutions on the right-hand side, a clear trend can still be seen.

To further examine what kind of Pareto optimal solutions are actually produced, we randomly chose one example run for closer observation. Figure 6 shows the final Pareto front of one example run of the 20 runs used in this experiment. In this example, Pareto optimal solutions are lying more on the efficiency side (five out of six solutions have a negative modifiability value), but the most modifiable solution still succeeds in achieving a modifiability value that is just as high as the highest efficiency value for the most efficient solution. In this run the Pareto front contained six solutions, while in the 20 test runs, the front could have anything between five and 13 solutions.

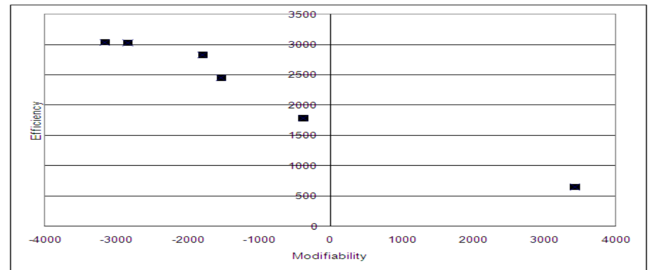


Figure 6. Pareto front of example run (after 250 generations)

To give an idea of the actual architectures in the Pareto front, let us study the architectures in the opposite sides of the Pareto front in the example run. In general, a central question in ehome architecture concerns the communication between the components. The mutual dependencies of components can be reduced by using message-based communication, increasing the modifiability of the system but decreasing efficiency. Other architectural solutions, in particular design patterns, affect in the same direction (that is, increase modifiability at the expense of efficiency), but they are clearly less dominating.

The most modifiable architecture of the example run is visualized in Figure 8. The sample architectures are depicted in a manually produced format which emphasizes the used solutions, not in the original UML class diagram format produced by the GA implementation, which would be too space-consuming and difficult to interpret. We give the architectures as component level presentations, where the pattern instances have been marked with short-hand notation.

The most interesting observation in the architecture of Figure 7 is the relatively extensive use of the message-based communication channel (Message dispatcher); in particular,

most of the communication between UI and device-specific components is proposed to be message-based. Note that the information concerning the expected frequency of certain service calls has affected the proposal. The architecture introduces also several basic modifiability patterns (Template Method and Strategy) for change-sensitive operations, and Adapters for change-sensitive interfaces.

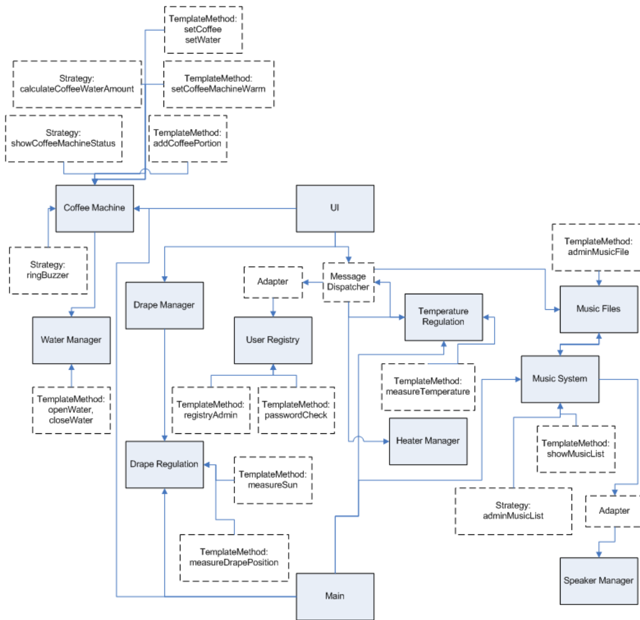


Figure 7. Modifiable solution

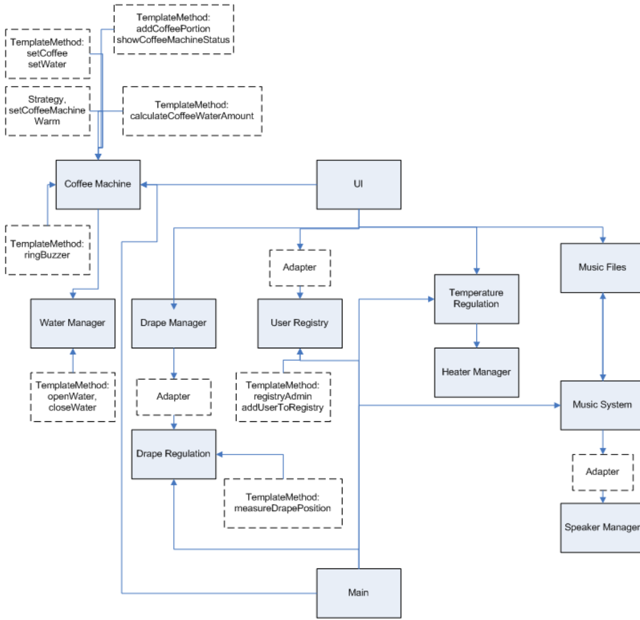


Figure 8. Efficient solution

At the other extreme, the most efficient proposal (that is, the architecture presented by the marker in the upper left corner in Figure 6) is shown in Figure 8 for comparison. In this case there is no usage of the message dispatcher and significantly less of the low-level patterns than in the previous proposal. Most of the used patterns are Template Method instances, which do not introduce notable efficiency cost.

Finally, one might also be interested in the "middle" solutions, i.e., ones that have neither an extremely high efficiency value, nor an extremely high modifiability value, but have a balanced fitness of both. In these solutions the message dispatcher is often present, but with only a few connections. The client-server architecture style is also used in some cases. A fairly moderate level of the usage of basic modifiability design patterns is usually proposed.

In practice, the designer is presented with a Pareto front scatter plot, such as in Figure 6. The designer can then select any point of the front from the plot, and the corresponding architecture is displayed. Thus, the designer is aware of what kind of fitness values each architecture has reached with the GA.

VI. EVALUATION

In this chapter we compare the results of our approach to the efficiency and modifiability assessment obtained from an imitated ATAM [7] evaluation. In ATAM, the stakeholders formulate scenarios which serve as test cases of certain quality goals. The target architecture is evaluated by analyzing to what extent the architecture supports the scenarios. Here we are in particular interested to see whether the quality distribution of the architectures in the Pareto front corresponds to the quality distribution of the architectures with respect to the imitated ATAM evaluation.

To carry out ATAM-like evaluation, we need to produce efficiency and modifiability scenarios. Efficiency scenarios can be produced in a relatively straightforward way. In ATAM, efficiency scenarios are typically use cases which involve those parts of the system that are particularly critical or otherwise interesting from efficiency viewpoint. Since the ehome system has five subsystems (user registry, coffee machine, music system, temperature control and drape control), we formulated five use cases which employ those subsystems, representing typical usages of the system. These use cases were then refined into sequences of operation calls.

Since we are here interested in the relative quality of the architectures rather than their absolute quality, it is sufficient to evaluate the efficiency of each architecture by computing relative time consumption for each efficiency scenario. We computed the total efficiency penalty of an architecture simply as the sum of the relative time consumptions of all the efficiency scenarios. In our evaluation, we used the following formula for the efficiency penalty *ep* of architecture x:

$$
\begin{aligned}
ep(\mathrm{x}) = & -\textstyle\sum \text{calls between different classes} \\
& - d^{*}\textstyle\sum \text{calls via dispatcher} \\
& - s^{*}\textstyle\sum \text{calls to server},
\end{aligned}
$$

where d and s are cost factors of message-based and client-server communications, respectively (relative to straight calls). In our experiments we set both d and s to 2. The exact values of these factors greatly depend on the way message dispatcher and server are implemented, which is beyond the scope of this paper. However, our tests indicate that using large coefficients would not essentially influence

the results of the evaluation; larger coefficients simply make differences between individuals larger, leading to more scattered fronts but retaining the relative order of the individuals.

Achieving modifiability scenarios is not straightforward, as building such scenarios requires intuitive understanding of the expected evolution of the system. To this end, we asked three software engineering experts (researchers with MSc or PhD not involved in our team) to construct four or five change scenarios each for the system as well as rough estimates of the likelihoods of the scenarios. They were given the functional requirements of the system as a basis. The experts produced in total 12 different scenarios that could be used for evaluation. This roughly corresponds to a real ATAM evaluation where typically 3-4 stakeholders produce 10-15 scenarios to be analyzed. An architecture was evaluated against a scenario by awarding 0, 1, or 2 points in the following fashion:

0 points: existing code would have to be changed
(no support),

1 point: existing code need not be changed,
but the architecture supports development time variation rather than run-time variation
(partial support)

2 points: existing code need not be changed
(full support).

The points were then multiplied with the probability of the scenario to achieve the total modifiability reward value for an architecture. In this experiment, the experts were not asked to evaluate the actual architectures. Results from architecture evaluations are given by Räihä et al. [18].

We used the final Pareto fronts of five different, randomly chosen, runs (including the example run given in Figure 6) from our experiment for the validation, each front having six, seven or eight different individuals. We evaluated how each individual in the Pareto fronts performed in the efficiency and modifiability scenarios, and gave penalty and reward points accordingly. The points of both quality attributes were then compared against how "high" in the Pareto front an individual was regarding that attribute. In other words, the most modifiable individual in the Pareto front should receive the highest points from modifiability scenarios when concerning individuals in the same front, and similarly the best in terms of efficiency (i.e., the one with worst modifiability) should receive the lowest penalty. Figure 9 shows the scatter plot for efficiency. Each Pareto front has a distinct marker in the plot. The y-value is the efficiency rank, i.e., number 0 is the first, and thus the most efficient, individual of the front (considering fitness values). Thus, the lower a marker, the closer to zero (small penalty) it should be on the x-axis.

Figure 10 gives a similar scatter plot for the modifiability scenarios. The markers here are the same as in Figure 9, i.e., the front represented by triangles in Figure 9 is represented by triangles in Figure 10 as well. Again, the individual with rank 0 is the most modifiable one of a given front, so the lower a mark, the higher its x-value (modifiability reward) should be.

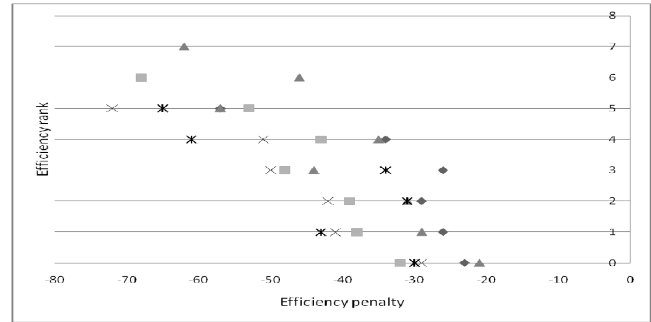


Figure 9. Efficiency scenarios

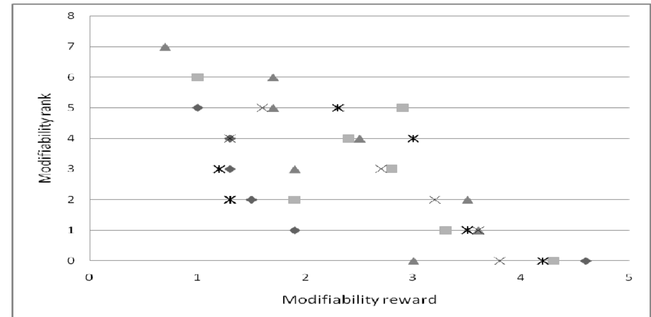


Figure 10. Modifiability scenarios

Performing statistical analysis on the scenario data produces correlation coefficient r values $r = 0.79$ for efficiency and $r = -0.62$ for modifiability. This indicates a high linear correlation between efficiency scenario points and the Pareto front rank, and a moderate correlation between the modifiability scenario points and the Pareto front rank. The corresponding coefficients of determination R^2 values are $R^2 = 0.62$ for efficiency and $R^2 = 0.52$ for modifiability, meaning that 62% of the variation in efficiency scenario penalty points can be explained by the Pareto rank, and similarly, 52% of variance within modifiability scenario reward points can be explained by the Pareto rank. Finally, the Student's t-tests for both data sets show that the impact of the rank in scenario points is significant with a 95% confidence (i.e., there is a significant correlation between rank and scenario points with $p < 0.05$), and thus the scenario points depend on the Pareto rank (i.e., the higher the rank, the better it performs in the scenarios).

Although the primary purpose of this work was to validate Pareto optimality in the context of genetic architecture synthesis rather than to evaluate the absolute "goodness" of the resulting architectures, it is interesting to observe that the best architectures got 7 modifiability points out of the maximum of 24 (without probabilities). Since the purpose of scenarios is to challenge the architecture, it is expected that a genetic process that is unaware of these scenarios will fail to produce specific solutions for many of them. In real life architectural evaluations we have been carrying out in industry, typical success rate for scenarios is 50-90%. Our results in this work are in line with those of Räihä et al. [18], suggesting that genetically synthesized architectures reach roughly the level of a third-year student.

We recognize that there are some threats involved in the validation. Firstly, the scenarios might represent too limited a view of real modifiability requirements – this is actually a potential weakness of ATAM itself. Secondly, we only use a limited amount of example runs for which the scenarios were evaluated. Thirdly, the example system might be biased towards our approach.

We argue that while the amount of used scenarios is limited, they do still give a sufficient enough picture of the demands for the whole system. For both quality attributes, there were one or more scenarios concerning each of the subsystems. In the case of efficiency, the use cases were also selected so that they were the most common, and thus the most critical ones. In the case of modifiability, already with this limited amount, the experts produced some duplicate scenarios (same scenario from several experts), which gives some confidence that the most crucial scenarios have been collected, and thus the impact of missing scenarios is not significant.

Similarly, the limited amount of test runs is not a problem, since the Pareto fronts of all the 20 runs clearly followed a similar structure, and the five runs used for validation were chosen randomly. Therefore, it is unlikely that the results would significantly change by adding or changing the runs used for scenario evaluation.

Finally, the example system is a fairly typical embedded system, consisting of UI, various controlling components, device drivers etc. As long as sensible architectures can be built using the available standard solutions, there is no obvious reason why the results would be essentially different for another system. Still, more experiments are clearly needed to confirm this.

VII. Conclusions and Future Work

Even though fully automated architecture design is beyond our approach, we see genetic synthesis as a promising research direction to develop an architect's tool that provides intelligent assistance by proposing possible designs that the architect can further elaborate. Such a tool can propose fresh solutions that a human designer, limited with her previous experience, could not even think of. This work suggests that the basic problem of conflicting goals in software architecture design, critical in genetic approaches, can be solved satisfactorily using Pareto optimality. We showed that the quality distribution of the architectures in the Pareto front is similar to the distribution obtained using imitated ATAM evaluation.

In our future work we plan on extending the collection of applicable patterns by implementing a way to introduce new patterns for GA without coding. We are also investigating applications of genetic architecture synthesis in the context of self-adapting systems, exploiting runtime information of the system's behavior. For such systems, the ability to automatically optimize the architecture according to dynamic information is a vital requirement.

VIII. Acknowledgements

The research was funded by the Academy of Finland (project Darwin). The authors would like to thank the software engineer experts who participated in the ATAM based evaluation.

IX. References

[1] A. Aleti, S. Björnander, L. Grunske, and I. Meedeniya, "ArcheOptrix: an extendable tool for architecture optimization of AADL models," In Proc. ICSE'09 Workshop MOMPES'09, 2009, pp. 61–71.

[2] M. Amoui, S. Mirarab, S. Ansari, and C. Lucas, "A genetic algorithm approach to design evolution using design pattern transformation," International Journal of Information Technology and Intelligent Computing 1, 2006, pp. 235–245.

[3] K. Beck, and W. Cunningham, "A laboratory for teaching object oriented thinking," ACM SIGPLAN Notices 24, 1989, 10, 1–6.

[4] M. Bowman, L. C. Briand, and Y. Labiche, "Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms," IEEE Transaction on Software Engineering 36, 6, 2010, pp. 817–837.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal, A System of Patterns – Pattern-Oriented Software Architecture., Wiley, 1996.

[6] S. R. Chidamber, C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering 20, 6, 1994, pp. 476–492.

[7] P. Clements, R. Kazman, M. Klein, Evaluating Software Architectures. Addison-Wesley, 2002.

[8] C. C. Coello Coello, "An Updated Survey of GA-Based Multiobjective Optimization Techniques," ACM Computing Surveys 32, 2, 2000, pp. 109–143.

[9] A. Diaz-Pace, H. Kim, L. Bass, P. Bianco, and F. Bachmann, "Integrating quality-attribute reasoning frameworks in the ArchE design assistant," In Proc. QoSA'08: Models and Architectures, LNCS 5281, Springer, 2008, pp. 171–188.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[11] M. Harman, A. Mansouri, and Y. Zhang, Search Based Software Engineering: A Comprehensive Analysis and Review of Trends, Techniques and Applications. Technical Report TR-09-03, Dept. of Computer Science, King's College London, 2009.

[12] M. Harman, and L. Tratt, "Pareto optimal search based refactoring at the design level," In Proc. GECCO'07, 2007, pp. 1106–1113.

[13] S. Kim, D.K. Kim, L. Lua, and S. Park, "Quality-driven architecture development using architectural tactics," Journal of Systems and Software 82, 8, 2009, pp. 1211–1231.

[14] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolutionary Programs. Springer, 1992

[15] O. Räihä, "A survey on search-based software design," Computer Science Review 4, 4, 2010, pp. 203–249.

[16] O. Räihä, K. Koskimies, and E. Mäkinen, "Genetic synthesis of software architecture," In Proc. SEAL'08, LNCS 5361, Springer, 2008, pp. 565–574.

[17] O. Räihä, K. Koskimies, E. Mäkinen, and T. Systä, "Pattern-based genetic model refinements in MDA," Nordic Journal of Computing 14, 4, 2008, pp. 338–355.

[18] O. Räihä, Hadaytullah, K. Koskimies, and E. Mäkinen, "Synthesizing architecture from requirements: a genetic approach," In P.Avgeriou, J. Grundy, J.G. Hall, P. Lago, and I. Mistrik (eds.): Relating Software Requirements and Architecture, Springer, Ch 18, 2011, to appear.

[19] M. Shaw, and D. Garlan, Software Architecture – Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[20] C.L. Simons, and I. C. Parmee, "A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design," Engineering Optimization 39, 5, 2007, pp. 631–648.