TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

NGUYEN KHAC HIEU
REVIEW OF SYSTEM DESIGN FRAMEWORKS

Master of Science thesis

Examiner: Prof. Timo D. Hämäläinen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 9th September 2015

ABSTRACT

NGUYEN KHAC HIEU: Review of System Design Frameworks
Tampere University of Technology
Master of Science thesis, 88 pages
September 2015
Master's Degree Programme in Electrical Engineering
Major: Wireless Communication Systems and Circuits
Examiner: Prof. Timo D. Hämäläinen
Keywords: System design framework, SoC, Methodology, Synthesis

In the last decade, the enormous development of the semiconductor industry with ever-increasing complexities of digital embedded systems and strong market competition with fast time-to-market and low design cost demands have imposed serious difficulty to a conventional design method. Therefore, there emerges a new design flow named model-based system design, which is based on high-level abstraction models, heavy design automation, and extensive component reuse to increase productivity and satisfy the market pressure.

This thesis presents reviews of ten high level academic system design frameworks and tools that have been proposed and implemented recently to support the model-based design flow, namely System-on-Chip Environment (SCE), Embedded System Environment (ESE), Metropolis, Daedalus, SystemCoDesigner (SCD), xPilot, GAUT, No-Instruction-Set Computer (NISC), Formal System Design (ForSyDe), and Ptolemy II. These tools are then compared to each other in various aspects comprising objective, technique, implementation and capability. Following that, three design flow frameworks, including ESE, Daedalus, and SystemCoDesigner, are experimented for their real usage, performance and practicality.

The frameworks and tools implementing the model-based design flow all show promising results. Modelling tools (ForSyDe, and Ptolemy II) can sufficiently capture a wide range of complicated modern systems, while high-level synthesis tools (xPilot, GAUT, and NISC) produce better design qualities in terms of area, power, and cost in comparison to traditional works. Study cases of design flow frameworks (SCE, ESE, Metropolis, Daedalus, and SCD) show the model-based method significantly reduces developing time as well as facilitates the system design process. However, most of these tools and frameworks are being incomplete, and still under the experimental stage. There still be a lot of works needed until the method can be put into practice.

PREFACE

The work of this Thesis has been carried out at the Faculty of Computing and Electrical Engineering, at Tampere University of Technology in Finland.

First and foremost, I would like to gratefully thank my supervisor *Prof. Timo D. Hämäläinen* for giving me an opportunity to work on the topic, and for his patient guidance and invaluable feedback during my thesis study.

Many thanks to all my friends here in Tampere who have spent more than two memorable and enjoyable years with me. Thank you for your help, sharing and precious friendship.

Finally, I also would like to express my profound gratitude to my parents and my special one for their love, support and understanding during my Master study.

Tampere, September 2015

Nguyen Khac Hieu

TABLE OF CONTENTS

1. Introduction . . . 1
  1.1 Background . . . 1
  1.2 Thesis Outline . . . 2
2. System-on-Chip Design . . . 4
  2.1 System Design Terminology and Concept . . . 4
  2.2 Development of System Design methodology . . . 5
  2.3 Model-based design methodology . . . 8
    2.3.1 Application modeling . . . 9
    2.3.2 Platform Definition . . . 9
    2.3.3 Mapping . . . 10
    2.3.4 Evaluation and Refinement . . . 11
    2.3.5 Implementation . . . 12
3. System design framework review . . . 14
  3.1 The System-on-Chip Environment . . . 14
  3.2 Embedded System Environment . . . 21
  3.3 Metropolis . . . 26
  3.4 Daedalus . . . 28
  3.5 SystemCoDesigner . . . 34
  3.6 No-Instruction-Set Computer . . . 38
  3.7 xPilot . . . 43
  3.8 GAUT . . . 45
  3.9 Formal System Design . . . 46
  3.10 Ptolemy II . . . 49
4. System design framework comparison . . . 54
  4.1 Comparing metrics . . . 54
  4.2 Tools and framework comparison . . . 55
5. Experimentation . . . 66

5.1 ESE 66
5.2 Daedalus 71
5.3 SystemCoDesigner 75

6. Conclusions 81

Bibliography 83

LIST OF FIGURES

2.1 System design methodologies 6
2.2 System design schedules 7
2.3 Model-based design steps 8
2.4 An example application model 9
2.5 An example platform model 10
2.6 An example mapping description 11
2.7 A system TLM 12
2.8 A cycle-accurate implementation 13

3.1 System-on-Chip Environment GUI 14
3.2 SCE general design flow 15
3.3 SCE architecture 15
3.4 SCE refinement-based design flow 17
3.5 Baseband example refined models 19
3.6 Baseband example Pin-accurate model 20
3.7 Embedded System Environment GUI 22
3.8 ESE Environment 22
3.9 ESE front-end 23
3.10 ESE back-end 23
3.11 MP3 Decoder and mapping decision 24
3.12 Comparison between manual design and ESE 24
3.13 Development time 25

3.14 Metropolis framework 26

3.15 Three processes communicate through a medium modeled by Metropolis metamodel 26

3.16 Architecture metamodel 27

3.17 Daedalus framework 28

3.18 Sesame model layers 29

3.19 ESPAM HW and SW implementation 31

3.20 Structure of CC and CB 32

3.21 Point-to-Point network implementation 32

3.22 Procees network model of Motion JPEG 33

3.23 Daedalus result comparison 33

3.24 SystemCoDesigner flow 34

3.25 Actor-oriented model 35

3.26 Actor HW implementation 35

3.27 Motion-JPEG block diagram 37

3.28 NISC design flow 38

3.29 NISC architecture 39

3.30 NISC control word 39

3.31 IP block diagram and its GNR description 41

3.32 CW with dictionary compression 42

3.33 xPilot design flow 44

3.34 Distributed register-file micro-architecture 44

3.35 ForSyDe system model 47

3.36 Basic process constructors 48

3.37 ForSyDe signal 48

3.38 Process implementation 49

3.39 A model in Ptolemy 49

3.40 Ptolemy GUI Vergil 50

3.41 The gas-powered generator top model 51

3.42 The SDF Controller model 51

3.43 The FSM Supervisor model 51

3.44 The Generator model 52

3.45 Result of the model 53

5.1 ESE GUI 67

5.2 JPEG encoder 68

5.3 JPEG encoder implementations in ESE 69

5.4 M1 statistic performance graphs 70

5.5 M3 statistic performance graphs 71

5.6 Sobel filter 72

5.7 Platform and mapping description 73

5.8 Sobel PPN visual representation 73

5.9 Statistic information 74

5.10 SystemCoDesigner main tools 76

5.11 H264 video codec 77

5.12 Total design point after 6 iterations 79

5.13 Latency, power, and area reports of Pareto design points 79

LIST OF TABLES

1.1 SoC Consumer Design Productivity Trends 1

3.1 Baseband modeling and simulation result [44]. 21

3.2 Baseband communication network 21

3.3 MP3 decoder generation and simulation time 25

3.4 TLM estimation error 25

3.5 Simulation and board implementation result 28

3.6 Resource utilization of design using 4 MicroBlaze processors 34

3.7 Processing Time 34

3.8 Design space exploration 38

3.9 Design space exploration 38

3.10 MicroBlaze and simple NISC 42

3.11 Comparison between MicroBlaze and simple NISC 43

3.12 Compasion between MicroBlaze, NISC-based MIPS, and NISC MP3 decoder customized datapath 43

3.13 xPilot Implementation reports 45

3.14 GAUT Implementation report 46

3.15 SystemC implementation of ForSyDe 49

4.1 Framework summaries 58

4.2 Framework resources 59

4.3 Comparison of purposes and targets 60

4.4 Comparison of architecture exploration and model accuracy 61

4.5 Comparison of case study 62

4.6 Comparison of model and design language 65

5.1 Design results 70

5.2 Feasible mappings of fundamental H264 video coder blocks 78

5.3 Attribute of PE components 78

5.4 Attribute of buses 78

5.5 Pareto design points mapping 80

LIST OF ABBREVIATIONS AND SYMBOLS

API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
CE	Communication Element
CPU	Central Processing Unit
CT	Continuous Time
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
DT	Discrete Time
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GUI	Graphic User Interface
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HIBI	Heterogeneous IP Block Interconnection
HW	Hardware
IC	Integrated Circuit
IP	Intellectual Property
KPN	Kahn Process Network
MoC	Model of Computation
MPSoC	MultiProcessor System-on-Chip
OS	Operating System
PE	Processing Element
PN	Processing Network
QoS	Quality of Service
RAM	Random Access Memory
RISC	Reduced Instruction-Set Computer
ROM	Read-Only Memory
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SDF	Synchronous Data Flow
SoC	System-on-Chip
SRAM	Static Random Access Memory
SW	Software

TLM	Transaction Level Model
VHDL	VHSIC Hardware Description Language
XML	Extensible Markup Language

1. INTRODUCTION

1.1 Background

Due to the rapid growth of semiconductor technology, especially in scaling feature of IC fabrication, and endlessly increasing demand of computation power, energy reduction, compactness, and reliability for implementing new complex applications, many systems has recently been built around a single chip called SoC, which includes highly integrated IP-blocks (multiple processors, HW accelerators, memory,...) and complicated interconnection. The system complexity, together with rising time-to-market pressure, now presents a great challenge, which is seemed infeasible for the current design techniques.

A 2011 report of the International Technology Roadmap for Semiconductors (ITRS)[4], shown in Table, predicted that the total logic size of SoC would be almost triple in this year 2015, in comparison to the year 2011, and could reach up to more than 38 times in 2026. The report also expected a percentage of design reuse would be 78% and 98% for year 2015 and 2026, respectively. Even with the enormous design reuse rate, the report showed that the productivity must be about 16 and 25 times higher in 2026 for new design and reused design to catch up to the manufacturing. This situation requires fundamental change in the design method, in which design should be carried out at higher abstraction level, and automation should be extensively utilized.

Table 1.1 *SoC Consumer Design Productivity Trends [4].*

Years	2011	2013	2015	2017	2019	2021	2023	2025	2026
SoC-CP Total logic size (normalized to 2011)	1	1.79	2.96	4.70	7.45	11.65	19.56	31.23	38.10
Required % of resuse design	54%	62%	70%	78%	86%	92%	95%	97%	98%
Require productivity for new design (normalized to 2011)	1	1.6	2.5	3.72	5.51	8.17	13.34	20.89	16.48
Require productivity for reused design (normalized to productivity for new design 2011)	1	1.6	2.5	3.72	5.51	8.17	13.34	20.89	25.24

Model-based design methodology has been introduced as a solution to the design challenges. In this design flow, application and platform are modeled at high-level abstraction level for analysis and exploration, then are gradually refined to lower-level implementation with pre-designed components. The methodology demands new, highly automatic frameworks and tools to efficiently bridge the gap between the high-level abstraction and the implementation. Numerous academic researches have been conducted on this, and several approaches have recently been proposed and implemented. Some present unified design environments, which cover the whole design flow from modelling, simulation, synthesis, or even verification ([44], [2], [7], [52]). Others focus on one specific aspect of the design steps, like high-level synthesis, which automatically generate HW and SW implementations from abstract models ([30], [16], [10]), or syntax and semantic of languages and rules for high-level modeling and simulation ([47], [23]).

This Thesis presents and compares ten promising academic high-level system design frameworks and tools, namely System-on-Chip Environment (SCE), Embedded System Environment (ESE), Metropolis, Daedalus, SystemCoDesigner (SCD), xPilot, GAUT, No-Instruction-Set Computer (NISC), Formal System Design (FSD), and Ptolemy II. The comparison is done on objectives, implementations, techniques, and capabilities of each framework and tool. Three design flow frameworks including ESE, Daedalus, and SystemCoDesigner are then chosen for experimentation to experience their real usage, performance and practicality. ESE is tested with JPEG encoder, Daedalus with Sobel filter, and SystemCoDesigner with H264 video codec. The thesis intends to provide a general view of currently existing high-level system design frameworks and tools, which can help designers choose the proper ones for their purposes as well as give developers ideas and directions for future tools development.

1.2 Thesis Outline

The second chapter provides background knowledge about system design. It introduces several commonly used terminologies and concepts, and presents a trend of design methodology to give a reason about the emergence of a new model-based design flow. The design steps of this new method are then given in details.

Chapter 3 shows reviews and summaries of ten academic system design frameworks and tools supporting the new model-based design flow, while Chapter 4 categorizes and compares them with several pre-defined metrics.

The following chapter presents the experimentation of three chosen design flow

frameworks, namely ESE, Daedalus, and SCD. It provides practical, detailed information about each framework structure, installation, and usages, and performance.

Chapter six gives a conclusion for the Thesis.

2. SYSTEM-ON-CHIP DESIGN

This chapter presents terminologies and concepts used in SoC design, developing trend of system design techniques, and a detailed design flow of the most modern design method - model-base design. The requirement of frameworks and tools support is also highlighted along the design steps.

2.1 System Design Terminology and Concept

The terminologies and concepts frequently used throughout the following chapters are listed below:

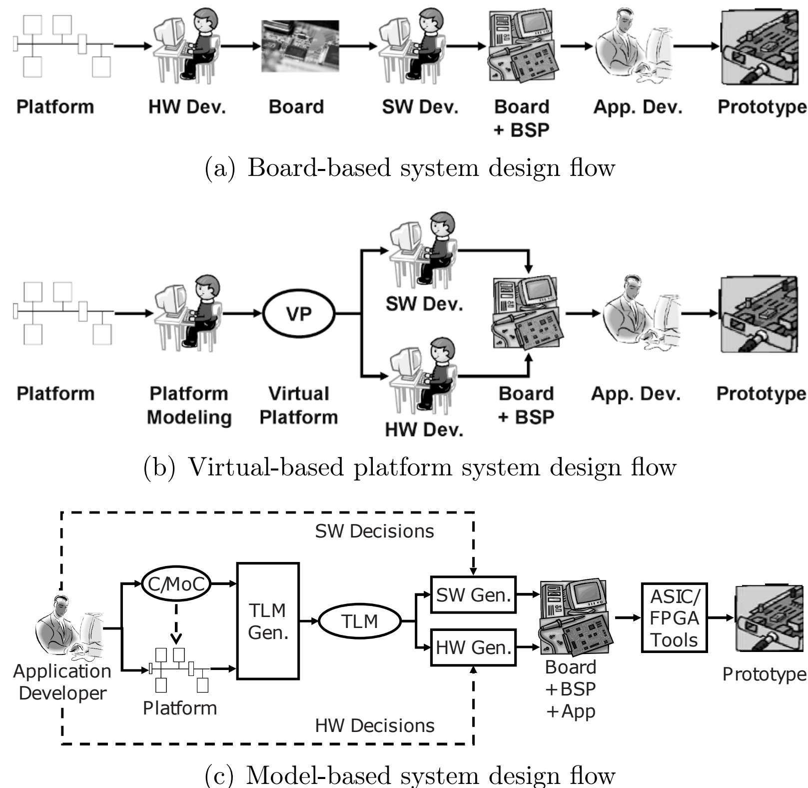
- **IP**: a pre-designed, pre-verified, reusable, and well-documented HW or SW design block with well-defined functionality and interface.
- **System**: an integrated entity of many interacting components to perform a predefined behavior.
- **System-on-Chip**: an electronic system formed by integrating distinct electronic components on a single chip. SoC contains HW IP-blocks and interconnections between them. IP-blocks can be processors, on-chip memories and memory controllers, interface for communication, Analog-to-Digital/Digital-to-Analog conversion, or accelerators (video encoder, graphics processor). Interconnection can include buses, bridges and routers.
- **Platform**: a SoC with specific HW IP-blocks and SWs that can be used to implement some applications. The platform can be changed in software (programmable platform) or hardware (configurable platform) to satisfy particular purposes.
- **Model of Computation**: There are several definitions of MoC [37, 39]. Basically, MoC is an abstracted, formal description of a system's behavior. It usually includes objects (pieces of behavior), and composition rules (object interactions and ordering of events). MoC has formal definitions and semantics for functionality, order of execution, and separation of computation and

communication. Common MoCs include KPN, Dataflow (DF), Finite State Machine, Statecharts, Discrete Event.

- **Mapping**: to specify implementation of an application or a part of its (tasks) on a particular ready-made component.
- **Synthesis**: to create an implementation by combining low level components or changing structure of a general purpose component.
- **Scheduling**: to specify orders and timings of tasks and communication executions.
- **Design Space Exploration**: all the possible combinations between component allocation, mapping and scheduling are analyzed regarding some selected objectives (cost, performance, power) to determine the most suitable design implementation.
- **Register Transfer Level**: a behavioral abstraction, in which time accuracy is clock cycle, system is described by register-transfer components, and communication unit is digital signal (0 or 1).
- **Transaction Level Model**: a behavioral abstraction, in which time accuracy is approximate timing, system is described by processes and channels, communication unit is data transfer event called transaction.

2.2 Development of System Design methodology

A system design flow consists of both HW and SW development with a repetitive sequence of specification, implementation, verification and testing processes. The traditional method depicted in Figure 2.1(a) is hardware-based approach. A platform is usually specified first with types of components and communication architecture regarding application features and desired requirements (performance, cost, power). Decisions made during this stage are mainly based on designer's experience and an application profile. The platform may be an update of a legacy design with added components to support the new application. The platform is then realized and verified by HW engineers at low RTL model to create a board, which is delivered to SW developers for board support packages (BSP) development (drivers, firmware). Finally, the application is ported into the board with BSP to create a product prototype. This design flow experiences long developing time due to a sequentiality and dependency of HW and SW developments, as shown in Figure 2.2. Any change or error occurs on the late design stage may involve modification of all



(a) Board-based system design flow

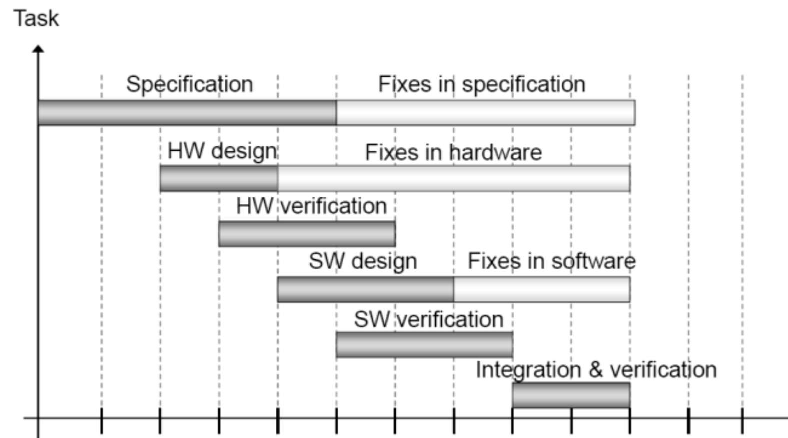
(b) Virtual-based platform system design flow

(c) Model-based system design flow

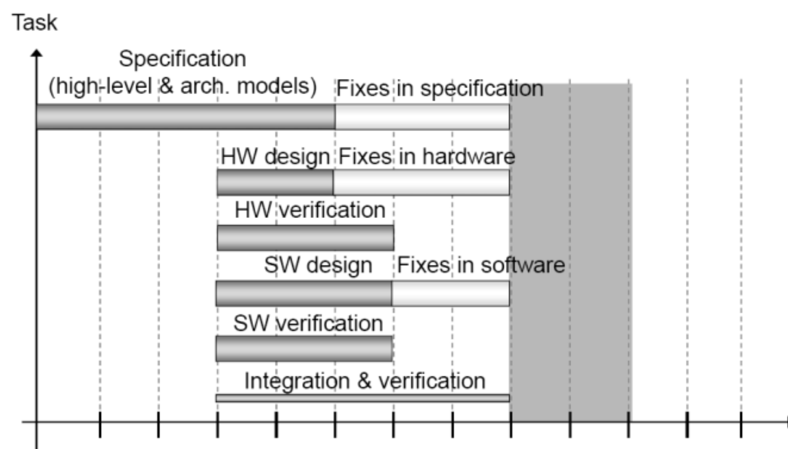
Figure 2.1 *System design methodologies [25].*

the previous stages, which cost enormous time and effort. Moreover, the separation between HW and SW cause poor interaction and understanding between the two teams, which can lead to wasted-time arguments and non-optimal final designs.

A solution to the shortcomings in the traditional board-based methodology is presented in a virtual-based platform design process. As illustrated in Figure 2.1(b), in this methodology, instead of a board, a model of platform called virtual platform is first developed and provided for SW development. Because the model is implemented at a higher level than RTL with many detailed platform features abstracted away, less time and effort are required to develop it than a concrete board. A virtual platform usually contains programmable models of processors and functional models of HW components. SW and HW can now be developed in parallel, which results in better interaction between them and shorter overall developing time. The virtual-based platform design flow, however, still contains a couple disadvantages. The additional work of virtual platform modeling as well as HW and SW development are mainly done manually, which is time-consuming and error-prone. Besides, there is still no quantitative evaluation method for platform definition, and newly rising multi-processor platforms may require old applications to be rewritten to fully take advantage of their computing power.



(a) Board-based design schedule



(b) Model-based system design flow

Figure 2.2 *System design schedules [35].*

To overcome these issues, a new design methodology introduced recently has completely re-defined the system design flow. The main idea of this methodology is to raise design work to higher abstraction level, and extensively exploit automation and design reuse to increase productivity and design quality.

The model-based methodology, depicted in Figure 2.1(c), employs both high-level models of functionality and platform for system development. An application modeled with appropriate MoCs is combined with a platform model to create an intermediate executable system model, usually TLM, which can be evaluated through simulation. The design decision can now be made with certain confidence based on a numerical result rather than experience. This evaluation model of a system also opens a capability of early design space exploration, in which many implementation alternatives can be estimated with much less time and effort. Well-defined semantic TLM can also be gradually refined down to final implementation with SW and HW generated from component libraries. Application modelling, TLM creation and re-

finement, and HW and SW generation all require supports from various high level frameworks and tools.

2.3 Model-based design methodology

The detailed design steps of model-based methodology, depicted in Figure 2.3, include 5 main processes,namely application modeling, platform definition, mapping, evaluation and refinement, and implementation.

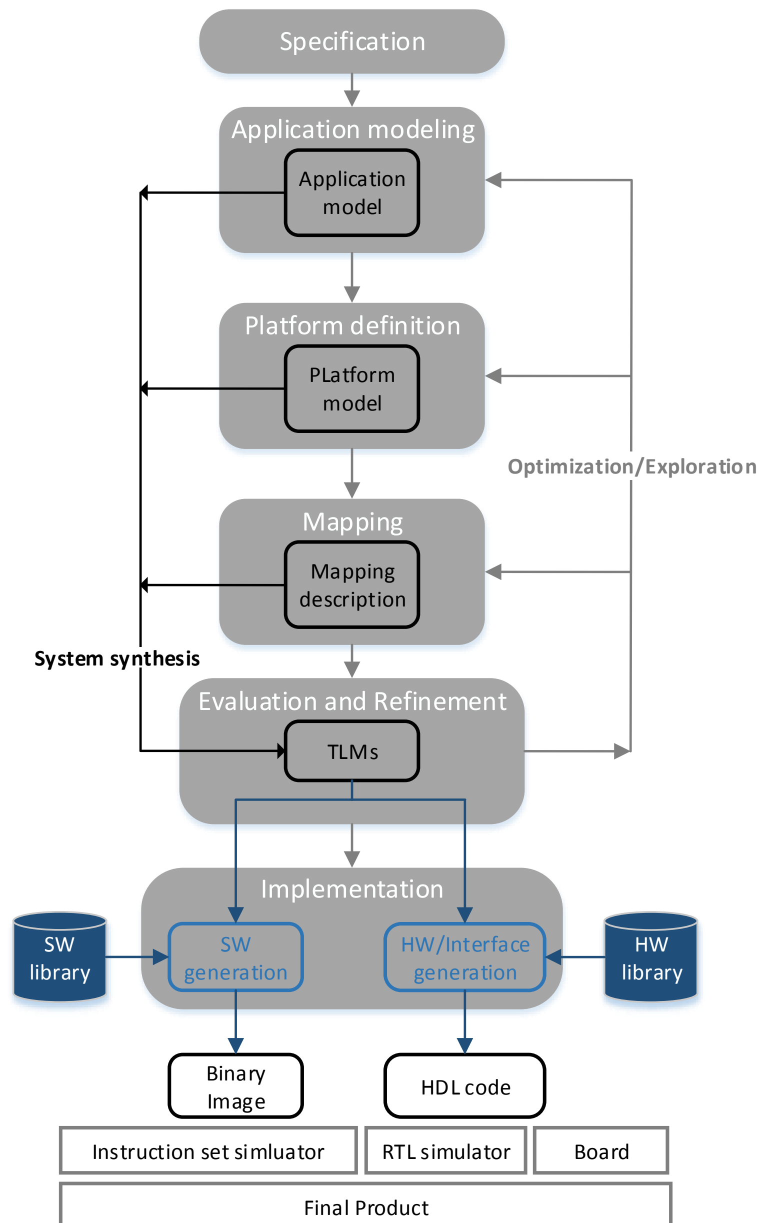


Figure 2.3 Model-based design steps.

2.3.1 Application modeling

The application modeling specifies a functionality of a system using well defined MoCs. A heterogeneous model with a combination of various MoCs is usually required to sufficiently describe real complex systems. Modeling languages are often C/C++ or its subsets and variants. The choice of MoCs and language mainly depends on the application domain, support of available design tools, and existing models. There is distinguish separation between computation and communication in the application model. Data processing units are described by *process*es, which transfer data to each other via *channel*s or shared variables. This separation is beneficial because it provides flexibility for distributing processes of an application on a multi-core platform, and enables independent development, modification, and optimization of computation and communication architecture. Application model is untimed. Sometimes, an application model is executed with instrumentation codes or a profiling tool to obtain computation timing estimation of each process, which is later served as a judgement for mapping decisions.

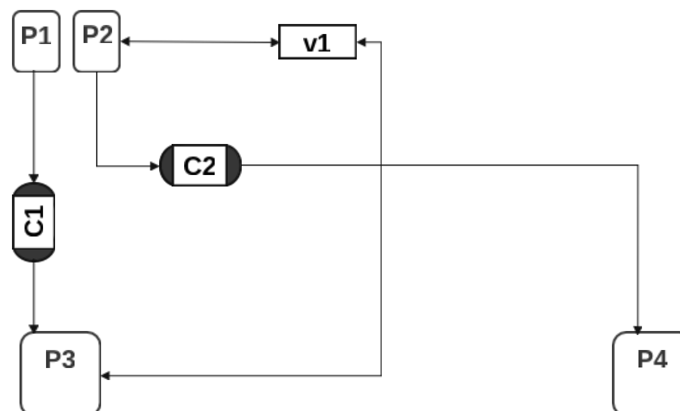


Figure 2.4 *An example application model described in Process State Machine (PSM) MoC with 4 processes P1, P2, P3, P4. Pairs of P1-P2 and P2-P4 communicate via channel C1 and C2, respectively, while P3 and P2 share data via variable v1 [35].*

2.3.2 Platform Definition

Platform definition is to choose applicable components, which provide services to carry out execution of a desired application. The components are HW parts (programmable processors, HW IP, buses, memories,...) and SW parts (OS) selected from libraries. Each component is often associated with high level model specifying functionality and several attributes such as computation or communication delay, area, power for mapping and evaluating. The platform model describes a structure of a system including allocated components and connections between them. It is

usually written in a declarative language like XML. The initial platform definition is usually simple and incomplete, and more detailed implementation will be added or generated through a sequence of decisions during the following developing stages.

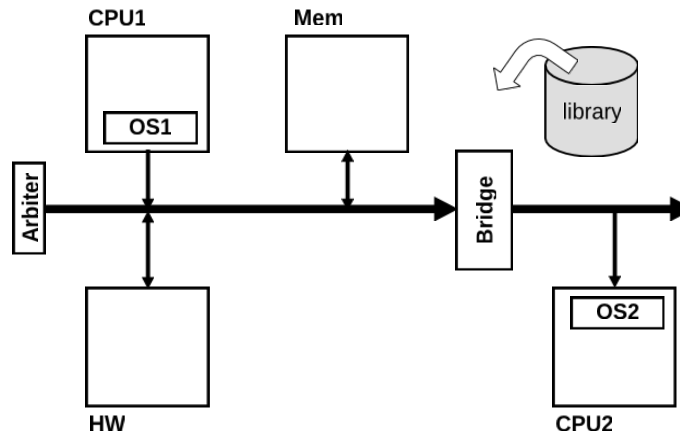


Figure 2.5 *An example platform model is composed of a memory, 2 processors CPU1, CPU2 and a HW as a PEs, two bus segments connected via a bridge, and a arbiter for controlling a shared bus. Each processor contains OS for multitask scheduling [35].*

2.3.3 Mapping

Application model and platform model must be combined to form an unified model of a system. The combination is specified by a mapping description, in which elements of the application model are partitioned into components of a platform model. Processes are mapped to PEs (programmable processors or HW IPs), channels are mapped to CEs comprised of buses, arbiters, bridges, and variables are mapped to memories. Multiple processes can be mapped into one processor with support of scheduling mechanism (OS or static scheduling), and multiple channels can be mapped to a shared bus with support of arbitration. Some mapping restrictions are usually taken into account during this step. A restriction can be feasible mapping components of each process, a restricted number of mapped processes of a processor, or a limited address space of a bus.

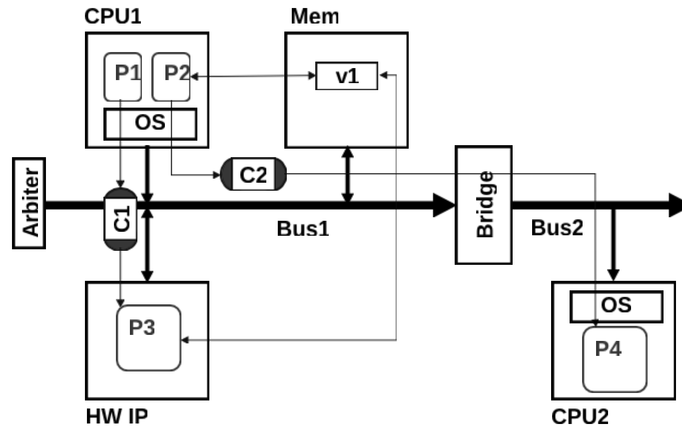


Figure 2.6 *An example mapping description, in which variable v1 is mapped into Mem, P1 and P2 are mapped into CPU1, P4 into CPU2, and P4 into HW IP. Channel C1 is mapped into a route based on shared bus 1, channel C2 into a route through bus 1 and bus 2 via bridges [35].*

2.3.4 Evaluation and Refinement

Given the application model, platform definition with associated component models, and mapping description, a high level executable model of a system - TLM - can be generated. It is a timing-estimated model, which provides a fast simulation speed while still maintains sufficient accuracy for system evaluation. TLM plays a main role in the model-based system design methodology. It provides the estimation of various metrics (performance, power, area) for system optimization and exploration, and is a main subject of system development.

Metric estimation of a TLM is compared against specified system requirements. These typically are real time constraints of automotive, and streaming multimedia systems, or power and area utilization constraints of portable devices. Any unsatisfied constraint requires modification back on previous design steps, such as choosing different components in platform definition, or changing the mapping scheme, or even editing application behavior. Due to high level developing environment, changes are effortless and quickly visible on the system model for further evaluation.

Platform definition and mapping process are often done manually, depending on experience of designers or legacy designs. However, due to the complexity of modern systems, the design space is usually enormous with thousands of feasible alternatives. Therefore, it is almost impossible to achieve the optimal design through brute-force work. There is a necessity for automatic tools, which can iterate the sequence of evaluation-allocation-mapping to figure out the best design solution. Those tools

usually employs heuristic algorithms, which use profiling of an application model and attributes of component models to generate Pareto-optimal design points regarding multiple desired constraints. Designers now have only few best choice for further investigation.

System development in model-based methodology is a sequence of TLM refinements. The latter model is refined from the previous ones, down to lower abstraction level with more detailed implementation, reflecting a more accurate model of a system. The TLM transformation mostly occurs at communication modeling with more networking layer added after each refinement.

TLM generation and refinement is expected to be implemented automatically in model-based methodology. Therefore, the semantics of TLM is required to be clear and well defined. Besides, this also can enable automatic synthesis from TLM to implementation with HW RTL description and compilable SW code, and formal verification through the whole design flow. TLM is commonly written in system level design languages like SpecC or SystemC.

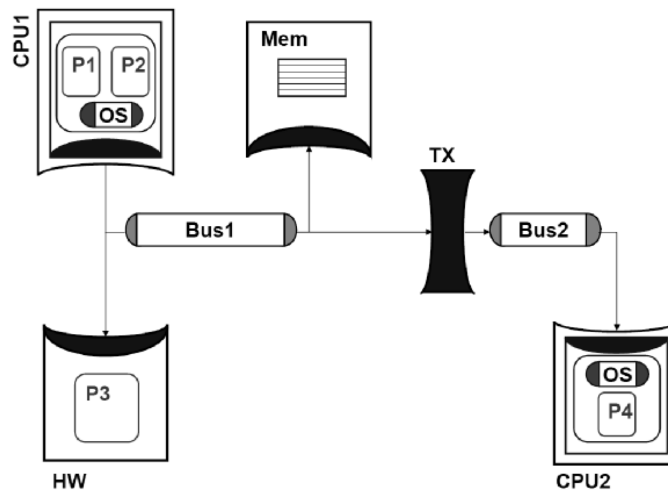


Figure 2.7 *A system TLM [35].*

2.3.5 Implementation

Final TLM will be synthesized to cycle-accurate implementation model. The synthesis process includes SW synthesis, and HW and Interface synthesis.

SW synthesis is composed of application code generation and HW-dependent code generation. The application written in system-level design languages of TLM is converted to C code. The suitable functions or data structures implementing data transfers between processes of application also are generated. HW-dependent SW

including drivers, RTOS, communication scheduling, control and synchronization mechanism are extracted from the SW libraries. HW-dependent SW generation also creates build and configuration files for correctly compiling and linking SW to the platform. All the codes are then compiled into binary images.

HW implementations are usually described in HDL codes. Some component are designed and verified in advance, and only needed to be allocated from the HW library. Others need to be generated on-the-fly by high-level synthesis tools, which convert an application behaviour in C into RTL description. HW synthesis also generate the connections and interfaces between components.

The generated binary images and HDL description are ready for final cycle-accurate simulation in an instruction set simulator and a RTL simulator, or to be implemented on a prototyping board. If everything is satisfied, a final product is released to the market.

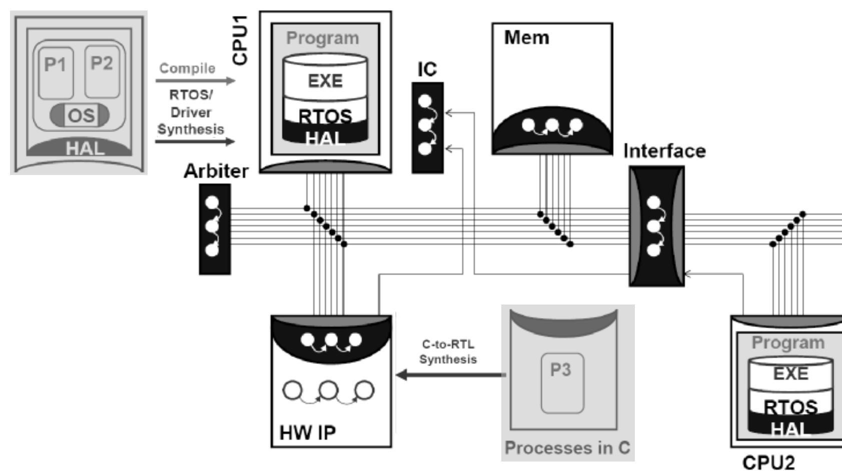


Figure 2.8 *A cycle-accurate implementation is synthesized from TLM [35].*

3. SYSTEM DESIGN FRAMEWORK REVIEW

The model-based design methodology requires support and automation of frameworks and tools for various tasks of its design steps. These include high-level modeling and simulation, system TLM synthesis, model refinements, high-level HW synthesis and SW generation. This chapter presents reviews of promising design frameworks and tools been recently developed in academia for the purpose of full realization of the model-based design flow.

3.1 The System-on-Chip Environment

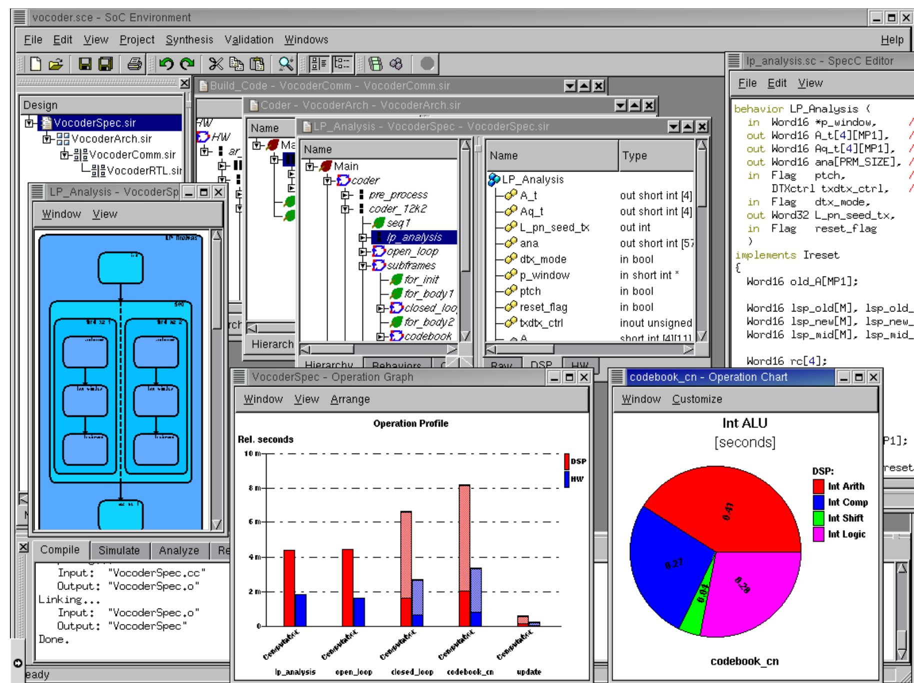


Figure 3.1 System-on-Chip Environment GUI [44].

The System-on-Chip Environment (SCE) is the system-level design framework developed at the Center for Embedded Computer Systems, University of California, Irvine. SCE follows the Specify-Explore-Refine principle, covering the communication and computation refinement steps of model-based system design flow.

As illustrated in Figure 3.2, starting with the specification model describing a system functionality written in SpecC, the SCE automatically generates a sequence of TLMs, namely architecture, scheduling, network and communication, based on various decisions made by users through successive design steps. The final TLM model is then synthesized to a cycle-accurate implementation model, which is composed of HW RTL descriptions in Verilog and target processor binary images of the SW.

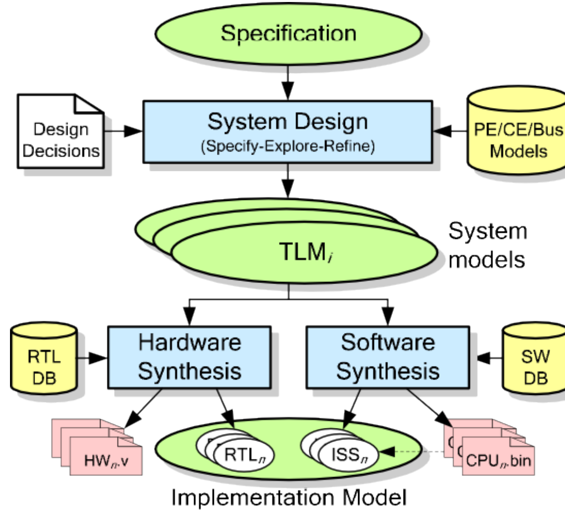


***Figure* 3.2** *SCE general design flow [44].*

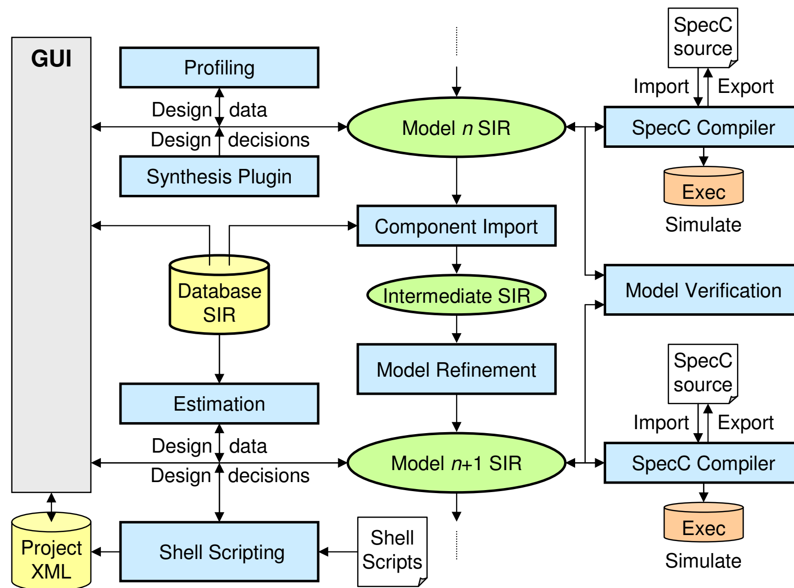


***Figure* 3.3** *SCE architecture [44].*

SCE architecture is described in Figure 3.3, including:

- Graphical User Interface: several displays, dialogs, menus and toolbars, which facilitate the design process as well as provide visuality for analyzing estimation result (Figure 3.1).

- Scripting Interface: own Python interpreter based shell with several command-line utilities, which provide full access and control to the SCE design flow with opportunities for automation via scripts or Makefiles.
- Simulation: all models in SCE are written in SpecC, which can be executed using SpecC compiler and simulator.
- Profiling: linear time re-targetable profiling tool, which supports multiple levels and metrics estimation. Profiling result can also be back-annotated into the output model of refinement.
- Databases: include PEs, CEs, operating system models, buses and other communication protocols, which are used for exploration and refinement stages. Custom IPs and SW components are also available for synthesis. All these components are SpecC objects.
- Model Generation and Refinement: several refinement tools (architecture refinement, OS refinement, network refinement, communication refinement, and SW and HW synthesis) are used to automatically generate output models reflecting the decisions made in the input models.
- Verification: formal Model Algebra based verification tool is available.

SCE implements the top-down refinement-based design flow from the abstract specification to implementation model, as illustrated in Figure 3.4. The design steps include:

- Architecture Exploration: an input is specification model. Components (processors, memories) are allocated from the databases to form a desired architecture. The processes and variables are then mapped to allocated PEs and memories. Channels are implemented in PEs as client-server, remote procedure calls. The output model is automatically generated by an architecture refinement tool.
- Scheduling Exploration: processes of processors are scheduled using abstract RTOS model written on top of SpecC. This supports task management, real-time scheduling, preemption, task synchronization, and interrupt handling. The output model is also generated automatically by a scheduling refinement tool.
- Network Exploration: network topology is defined, and communication channels are mapped to bus networks and allocated CEs (bridges, transducers,...).

A refinement tool generates the output model with communication protocol top layers in each PE and CE.

- Communication Synthesis: communication in the input model is a set of logical links. Bus parameters (address, interrupt assignment) are assigned by users in this step. A communication refinement tool generates an output model, in which low-level communication layers in PEs and CEs are inserted to implement synchronization, addressing, and media access. The output model can be pin-accurate model (PAM) or fast-simulating TLM.

- RTL Synthesis: all or parts of the design can be synthesized to RTL descriptions in Verilog. A RTL-specific profiling tool is also available for analyzing delay, power, variable lifetime.

- SW Synthesis: ANSI-C code generated from the high-level design language is compiled and linked to selected RTOS on each processor to form a binary image. The binary can be used for cycle-accurate instruction-set simulation or implementation on real HW.

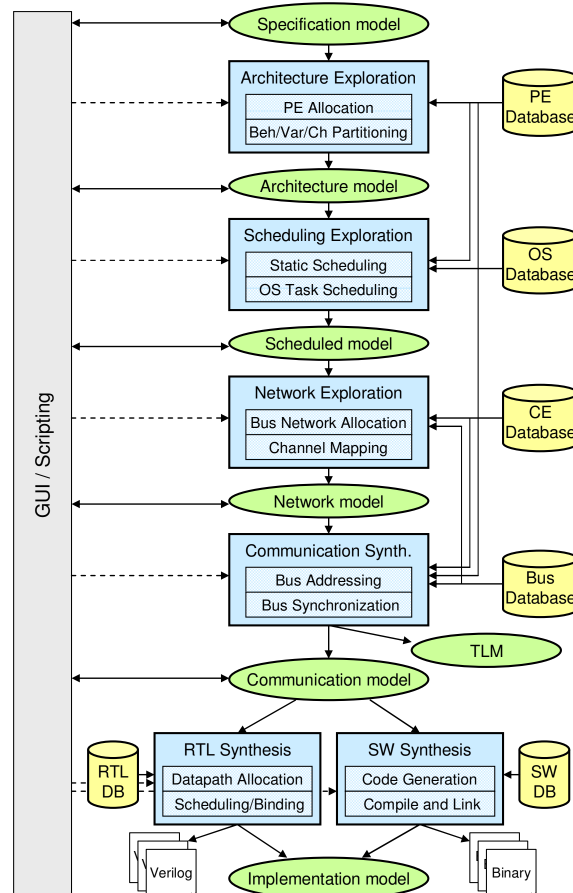


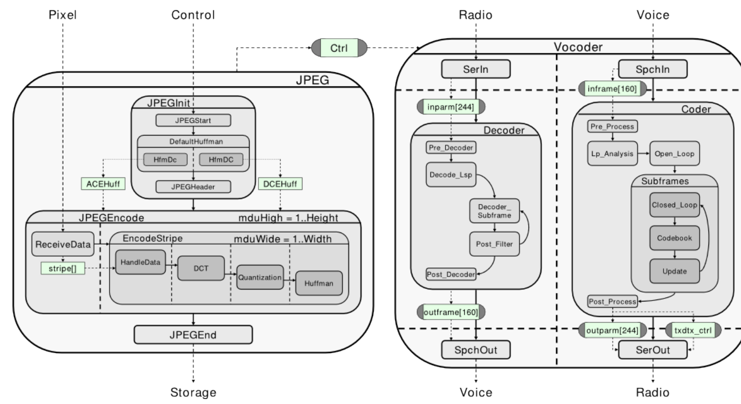
***Figure* 3.4** *SCE refinement-based design flow [13].*

SCE has been used to design a mobile phone baseband platform comprised of JPEG encoder and voice encoder/decoder (vocoder) subsystem [44]. The specification model of the system in Figure 3.5(a) describes two subsystems containing nested and pipelined task loops communicating via abstract message-passing channels. The JPEG encoder sends messages to the vocoder via the *Ctrl* channel.

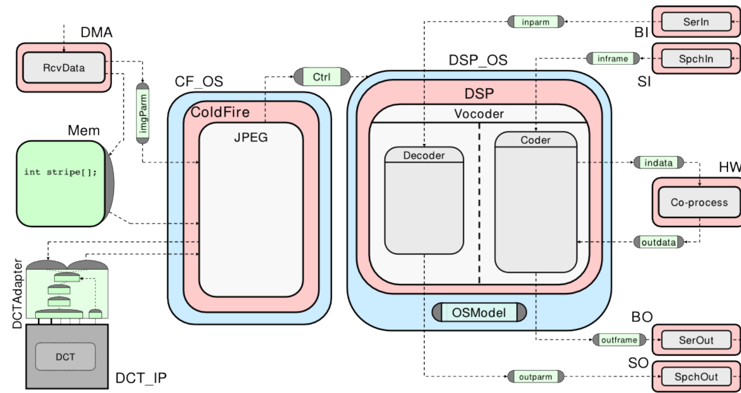
After Architecture Exploration and Scheduling Exploration steps, JPEG encoder is partitioned to the *ColdFire* processor and the HW IP running DCT sub-task, whereas concurrent speech encoding and decoding sub-tasks of the vocoder are implemented on the DSP with OS layer. Additional custom HW coprocessor performs heavy-processing codebook search sub-task of the encoder. Besides, there are DMA for transferring images from camera to shared memory *Mem*, and a custom HW I/O processor for buffering and framing vocoder speech and bit streams, as shown in Figure 3.5(b).

The network is divided into two separate segments in network exploration step, one for the JPEG encoder subsystem and one for the vocoder subsystem. These two segments are statically routed via the transducer *Tx - Ctrl*. The details of the network are listed in Table 3.2. The output models include TLM (Figure 3.5(c)) for fast simulating and pin-accurate model (Figure 3.6) for further implementation.

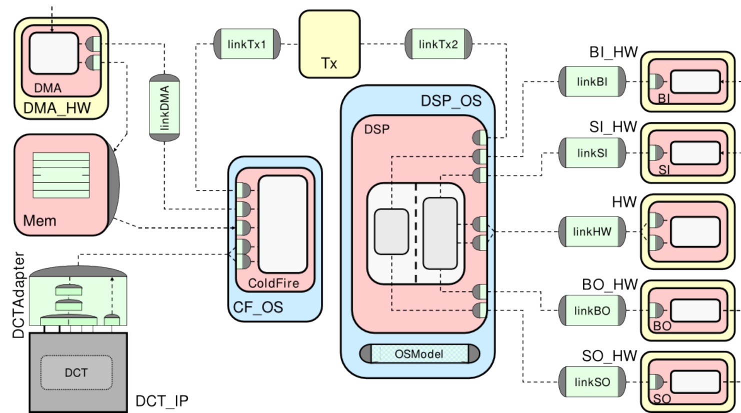
The result of the design is summarized in Table 3.1. Refined models are generated for a whole system, and for each subsystem separately. A test-bench contains simultaneously encoding and decoding 163 frames of speech while encoding 30 116x96 pixels JPEG pictures, and is simulated on 2.7GHz Linux workstation. The table shows that all of the models are generated using automatic refinement tools within only a few seconds, about 2.25 seconds on average. The accuracies of the models converge through the design flow. There is no significant difference between the performance from the scheduled model to the RTL-C model in both subsystems with about 3.6% at worst in the JPEG encoder and around 6.7% in the vocoder.



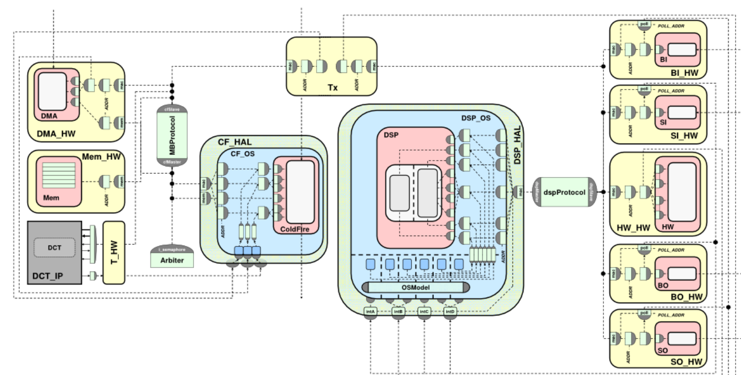
(a) Specification model



(b) Scheduled Architecture model



(c) Network model



(d) Transaction level model (TLM)

Figure 3.5 *Baseband example refined models [44].*

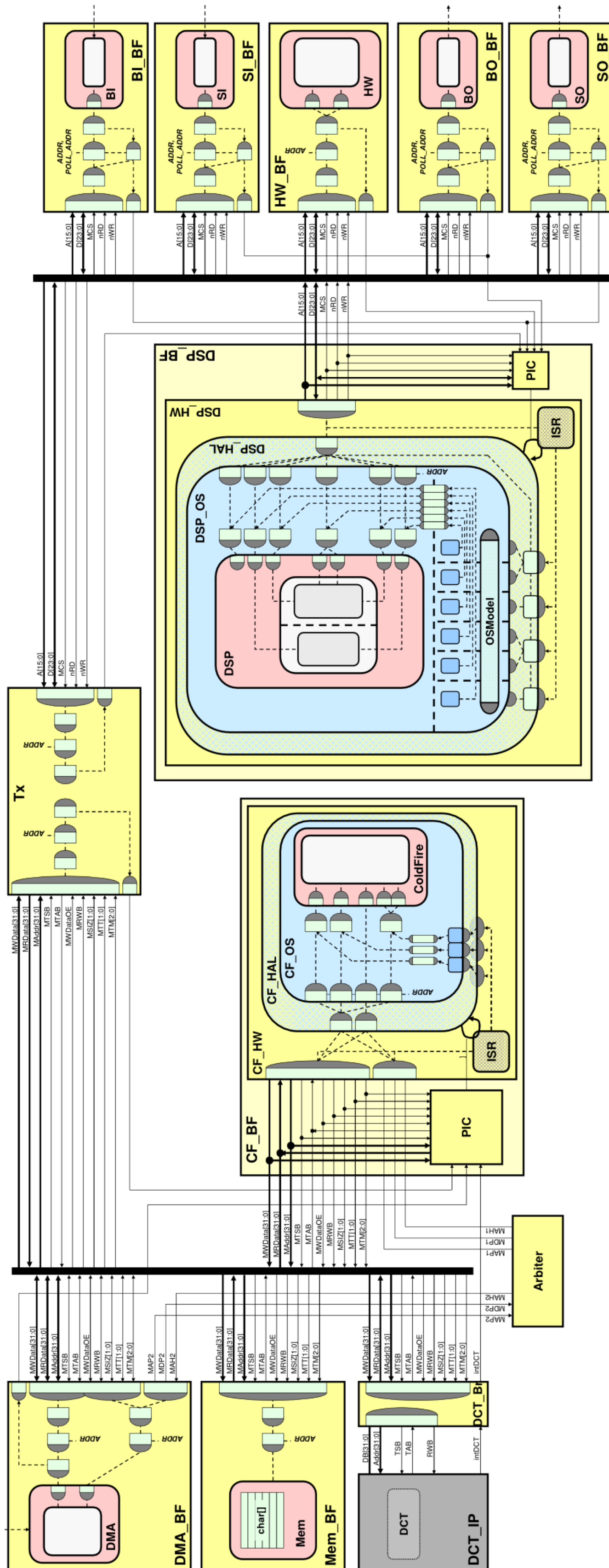


Figure 3.6 *Baseband example Pin-accurate model [44].*

Table 3.1 *Baseband modeling and simulation result [44].*

	ColdFire subsystem			DSP subsystem			System		
Model	LOC	Simul. time	JPEG delay	LOC	Simul. time	Vocoder delay	LOC	Simul. time	Refine. time
Specification	1,819	0.02s	0.00ms	9,736	1.31s	0.00ms	11,481	2.25s	
Architecture	2,779	0.03s	9.66ms	11,121	1.21s	8.39ms	13,866	2.56s	4.27s
Scheduled	3,098	0.02s	22.63ms	13,981	1.20s	12.02ms	17,020	2.00s	2.46s
Network	3,419	0.02s	22.63ms	14,319	1.22s	12.02ms	17,658	2.03s	1.24s
TLM	5,765	1.04s	24.03ms	15,668	27.4s	13.00ms	21,446	92.3s	1.02s
PAM	5,916	14.2s	24.02ms	15,746	34.8s	13.00ms	21,711	2,349s	
RTL-C	7,991	14.9s	23.48ms	23.661	147s	12.88ms	33,511	2,590s	

Table 3.2 *Baseband communication network [44].*

Channel	Network	Link		
	Routing	Addr.	Intr.	Medium
imgPam	linkDMA	0x00010000	int7	cfBus
stripe[]	Mem	0x0010xxxx	-	
hData	linkDCT	0x00010010	int1	
dData				
Ctrl	linkTx1	0x00010020	int2	
	linkTx2	0xB000	intA	dspBus
inframe	linkSI	0x800x	intB	
outparm	linkBO	0x950x		
indata	linkHW	0xA000	intD	
outdata				
inparm	linkBI	0x850x	intC	
outframe	linkSO	0x900x		

3.2 Embedded System Environment

Embedded System Environment (ESE) is a product of the Center for Embedded Computer Systems, University of California, Irvine, and being developed as a successor of SCE. ESE follows the model-based design methodology, supporting automatic TLM generation and HW and SW synthesis. ESE is composed of two main parts, namely the front-end and the back-end, as illustrated in Figure 3.8.

The ESE front-end detail described in Figure 3.9 generates automatically SystemC TLM from system specification for fast and early design evaluation. The input application is a set of concurrent C/C++ processes communicating to each other using

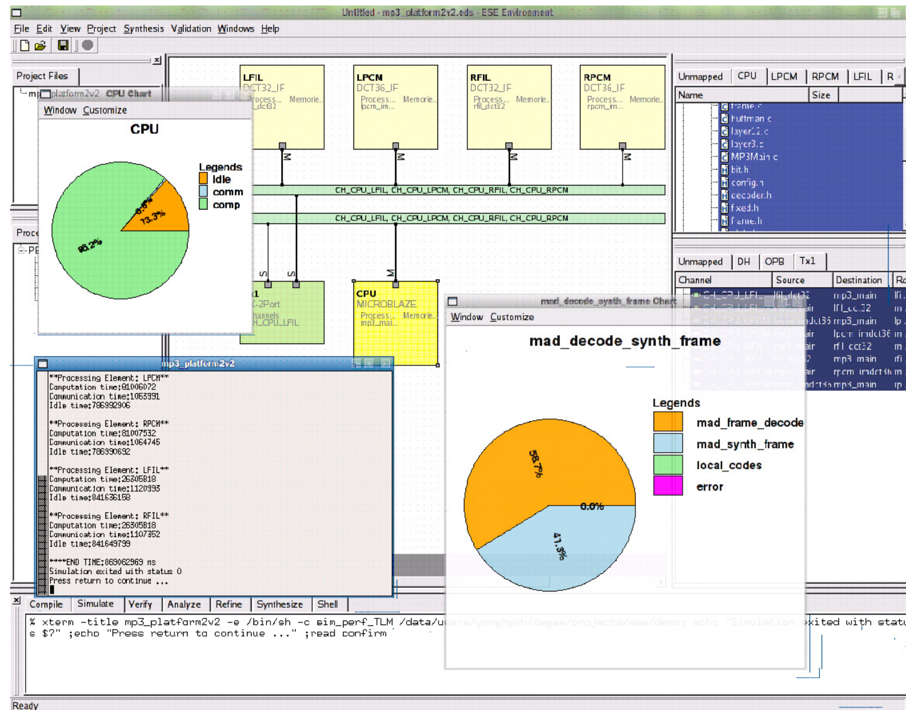


Figure 3.7 Embedded System Environment GUI.

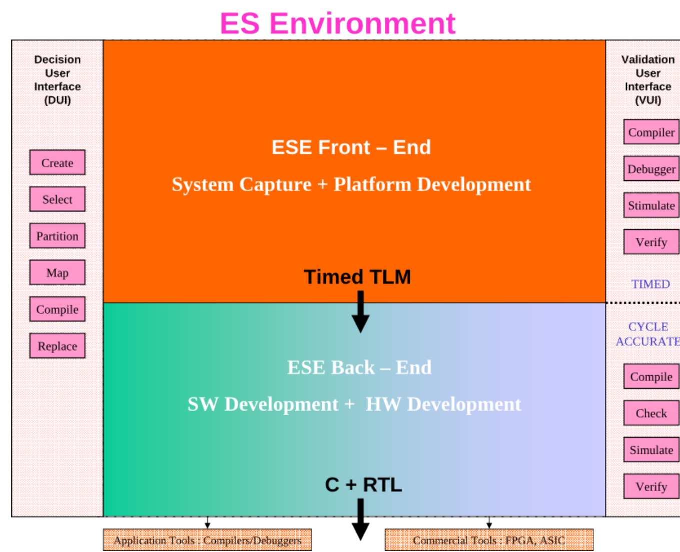


Figure 3.8 ESE Environment [21].

simple methods *read(data ptr, data length)/write(data ptr, data length)* of which detailed implementations are created by ESE. The processes and communicating methods are mapped to allocated PEs and configured channels, respectively. A PE can be a processor, a pre-designed IP, or a custom HW model, which lately can be synthesized using third party high-level synthesis tools (NISC, Forte). A channel can be chosen amongst FIFO, asynchronous or synchronous transfer. RTOS model can be selected in a processor to handle task scheduling. Component allocation and platform mapping are done using the GUI. ESE uses LLVM operations to estimate

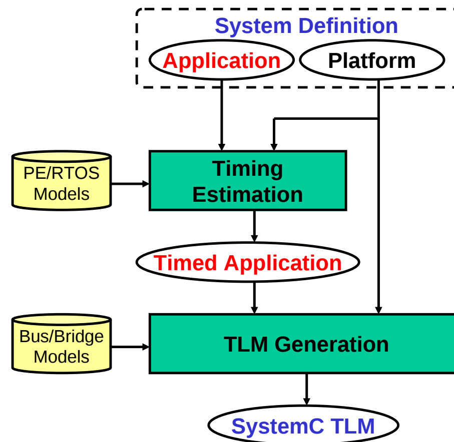


Figure 3.9 *ESE front-end [25].*

execution time of the application [1, 34]. A PE model contains delay cost of each LLVM operation along with a data-path, and stochastic delay of memory access and branch prediction. The LLVM-compiled version of a program code is used for computation timing estimation, which then is annotated into the processes to create timed application model. The timed application model then is combined with transaction delays of bus models, scheduling delays and inter-process communication delays of OS models to generate the final SystemC timed TLM (TTLM). In TLM, the application is modeled as *sc_thread*, OSes are modeled as *sc_modules*, which support several POSIX methods, the PEs are modeled as *sc_modules*, buses are modeled as *sc_channels* using Universal Bus Channel (UBC) template [2], which provides functions for routing, synchronization, arbitration and data transfer, memories are modeled as arrays inside *sc_module*, and bridges are modeled as FIFO channels *sc_process*.

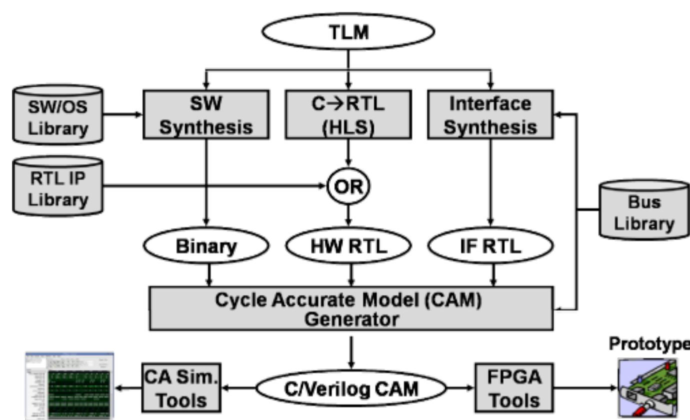


Figure 3.10 *ESE back-end [25].*

The ESE back-end includes SW synthesis, HW synthesis (C-to-RTL) and Interface synthesis, as shown in Figure 3.10. In SW synthesis, a program is compiled to specific target processor using chosen compiler and library. RTOS model is replaced by

selected RTOS. Address range for SW programs and data memories are also assigned. HW models are replaced by pre-designed IPs or synthesized to Verilog by high-level HW synthesis. In Interface synthesis, synchronization in UBC is implemented using polling, a CPU interrupt, or an interrupt controller, arbiters and bridges are automatically-synthesized or extracted from library, communications in SW are replaced by RTOS functions, and communications in HW are implemented using generated DMA controllers. The final result of the ESE is the CAM with SW binary images and HW Verilog RTL codes, which are ready for simulation or prototyping.

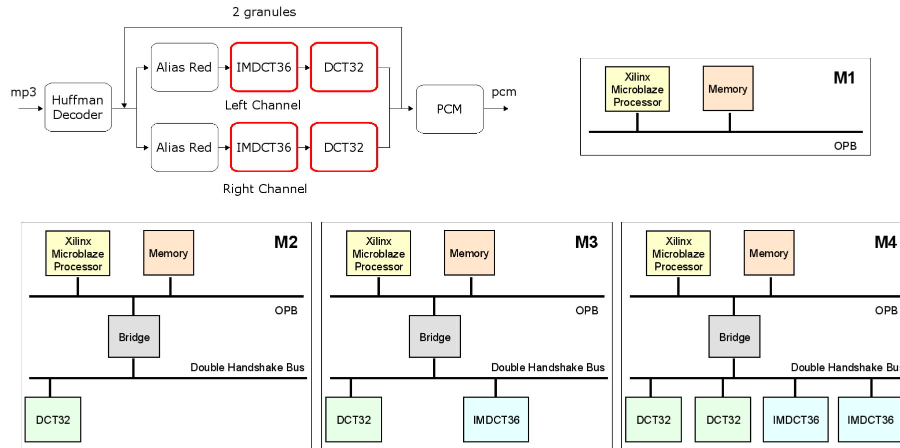
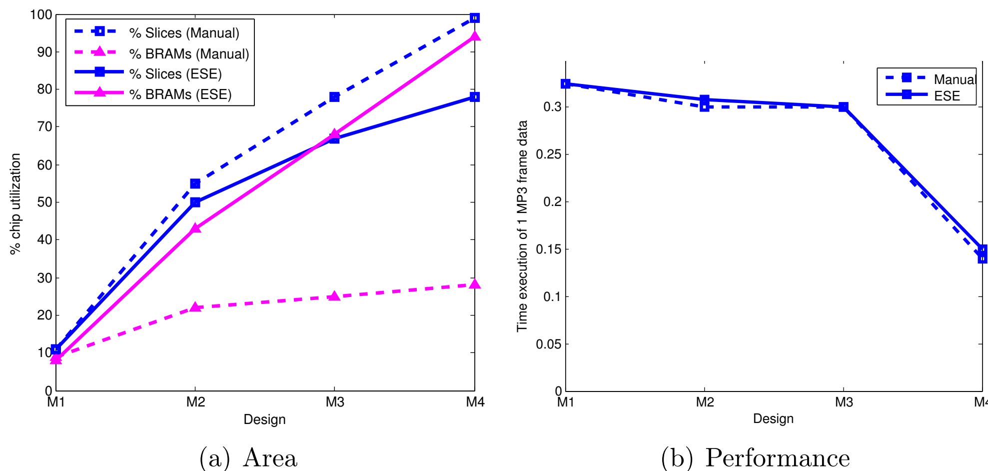


Figure 3.11 *MP3 Decoder and mapping decision [1, 2, 3].*

MP3 decoder is used as a case study in ESE [1, 2, 3]. Its general processes are described in Figure 3.11 with two intensive computing tasks IMDCT36 and DCT32. Four platform alternatives are chosen, including pure SW implementation on the Microblaze processor M1, Microblaze and custom left channel DCT HW M2, custom left DCT and IMDCT HW M3, and custom left and right DCT and IMDCT HW M4. Generation and simulation time of SystemC TLMs are described in Table 3.3.



(a) Area (b) Performance

Figure 3.12 *Comparison between manual design and ESE [21].*

The designs are automatically generated in tens of seconds. The TLM simulation time is also in order of seconds in comparison with couple of hours in cycle-accurate simulation. The accuracy comparison results between board prototype and TLMs with various cache configurations are shown in Table 3.4, in which the average error is only around 7% and around 19% in the worst case. It also shows the effect of different configurations of SW and HW models on the accuracy of generated TLMs. The final implementation result on FPGA board in Figure 3.12 shows that ESE consumes fewer FPGA slices and more BRAMs than the manual design, and there is no significant difference in execution time between these two. ESE shows comparable design quality as the manual work, while provides remarkable productivity gain in development time, as illustrated in Figure 3.13.

Table 3.3 *MP3 decoder generation and simulation time [2].*

Design	Timed TLM Generation	Timed TLM Sim	CA Sim
M1	31s	1s	16h
M2	50s	22s	18h
M3	47s	25s	18h
M4	71s	36s	18h

Table 3.4 *TLM estimation error [2].*

Cache Size	M1	M2	M3	M4
0K/0K	6.27%	9.00%	18.18%	18.61%
2K/2K	6.68%	-7.16%	-15.79%	-9.35%
8K/4K	4.74%	9.13%	-1.66%	-0.18%
16K/16K	-13.83%	4.66%	2.63%	3.65%
32K/16K	-13.89%	-8.29%	1.57%	2.29%

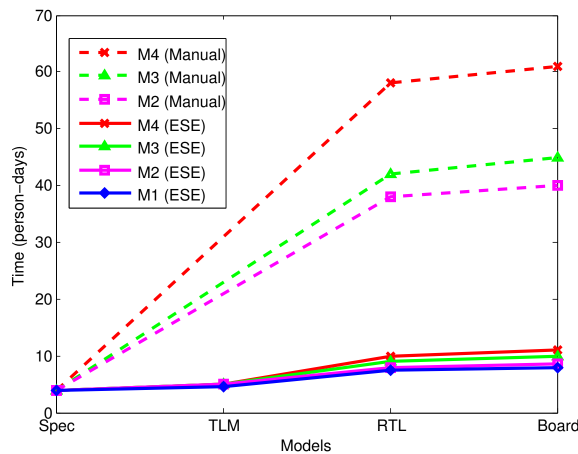


Figure 3.13 *Development time [21].*

3.3 Metropolis

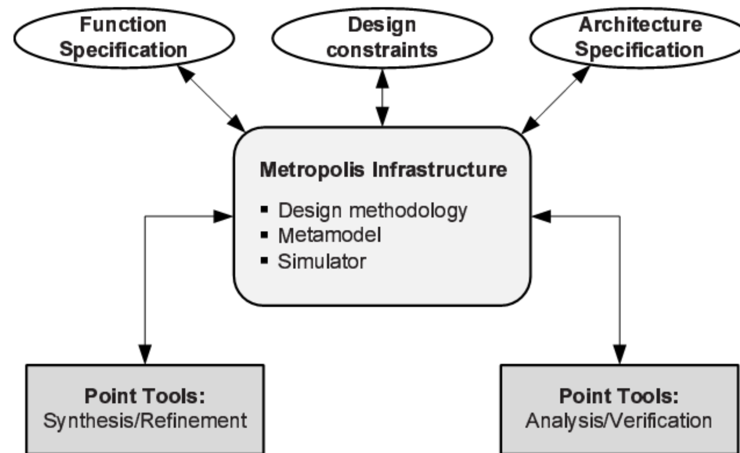


Figure 3.14 *Metropolis framework [25].*

Metropolis is the platform-based design framework developed by University of California, Berkeley in collaboration with several other universities and companies. Metropolis includes metamodel language for functionality and architecture specification, and set of simulation, and verification tools, as shown in Figure 3.14.

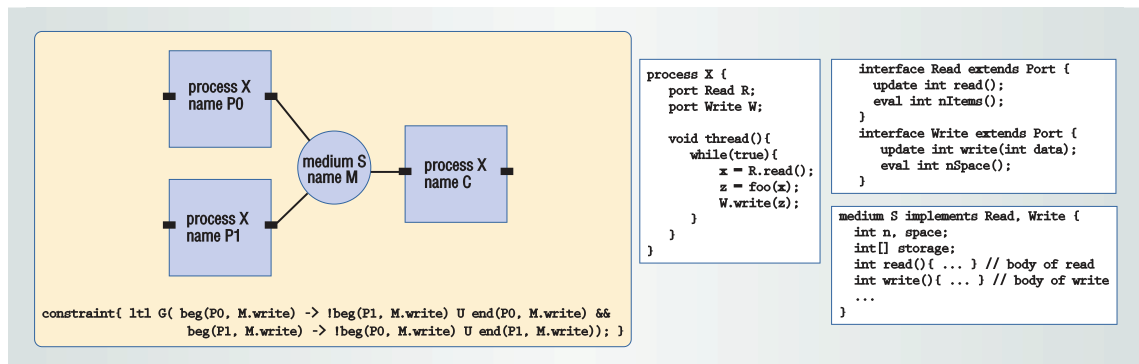


Figure 3.15 *Three processes communicate through a medium modeled by Metropolis metamodel [7].*

In Metropolis metamodel, system functionality is described as a concurrent process, which implements sequential programs called threads. Processes communicate to each other via port interfaces connected through objects called media, as shown in example in Figure 3.15. The communicating methods are declared in port interfaces and defined in the media, which allows any media with compatible methods can connect to the ports. This shows the separation between communication and computation in the metamodel. The architecture is modeled like the functionality, but with additional objects called quantity managers. A quantity manager manages the

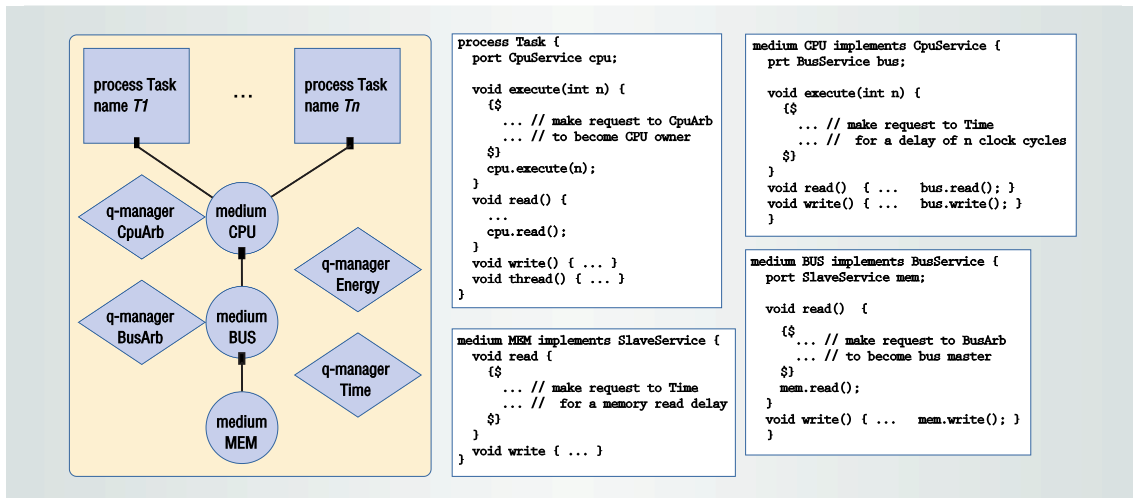


Figure 3.16 *Architecture metamodel [7].*

access to shared resources of the architecture (CPU, bus,...) , and annotate several performance quantities such as time, power,... Simple example of an architecture model is shown in Figure 3.16.

The metamodel specification can be converted to Abstract Syntax Tree, which then is synthesized to SystemC language for simulation. Formal verification including Linear Temporal Logic (TTL) and Logic of Constraints (LOC) are supported with LOC checker and SPIN model checker. HW synthesis can be implemented using the third-party xPilot synthesis tool.

Metropolis 2 has been currently under development with three main improved features, namely pre-designed IP importation, non-functional and behavior separation, and structured design space exploration. GUI running under Eclipse is also in progress.

One study case of Metropolis is Motion-JPEG (MJPEG) [22] implemented with MicroBlaze processor as a computation unit and Fast Simplex Link as a communication element. The application and architecture are modeled using Metropolis metamodel. The MJPEG is composed of Preprocessing (P), DCT (D), Quanatization (Q), Huffman Encoding (H), and Table Modification (TM) tasks, which are partitioned in 4 different platform models. The result in Table 3.5 shows the execution time difference between simulation and real board implementation. The difference is about 8% on average and 25% at worst. Area and max frequency from synthesis are also reported.

Table 3.5 *Simulation and board implementation result [9].*

Model	Simulation Cycles	Real Cycles	Max MHZ	Execution Time	Area (slices)
M1	228356 (25%)	304585	101.5	0.0030s	4306
M2	145659 (6%)	154217	72.3	0.0021s	4927
M3	145414 (1.2%)	147036	56.7	0.0026s	7035
M4	144432 (<1%)	143335	46.3	0.0031s	9278

3.4 Daedalus

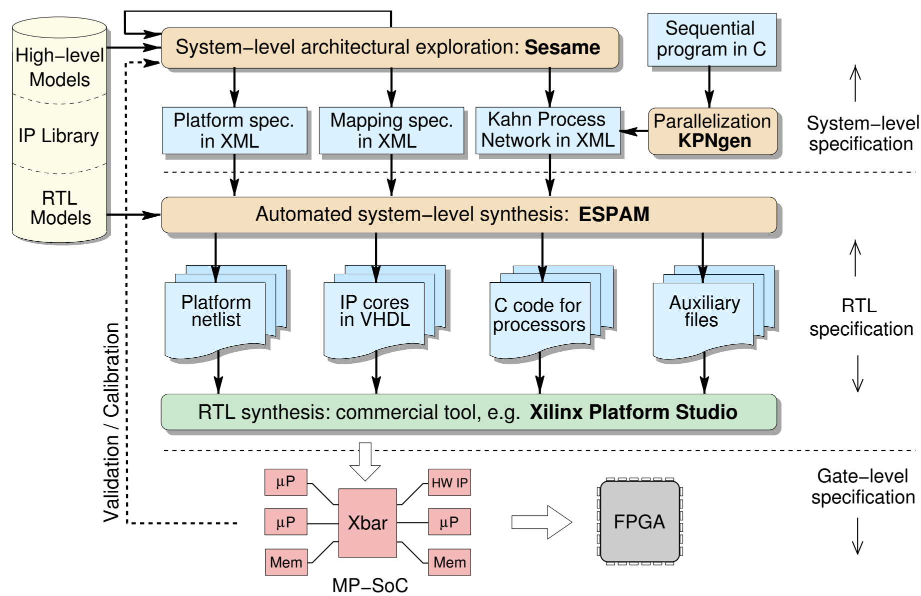


Figure 3.17 *Daedalus framework [41].*

Daedalus is an integrated tool-flow environment developed under cooperation between University of Amsterdam and Leiden University, focusing on streaming multimedia application. The design flow of Daedalus illustrated in Figure 3.17 comprises of three separate main tools, namely PNgen, Sesame and ESPAM, which together facilitate and automate the system design process.

The PNgen [57] is a parallelization tool based on Compaan [38, 46, 54] research project of systematic and automatic network process derivation. It efficiently converts a sequential application to a parallel process network, which offers effortless, apparent and potentially automatic mapping to heterogeneous and multiprocessor platforms. The input of PNgen is C/C++ application constrained in form of Static Affine Nested Loop Programs (SANLP), and the output is the special subset of KNP model named Polyhedral Process Network (PPN). PNgen is optimized from Compaan to achieve potentially fewer channels and processes, potential replacement of

reordering channels with FIFO channels, and compiled-time FIFO size determination. Different equivalent PPNs with various levels of parallelism can be generated for design space exploration.

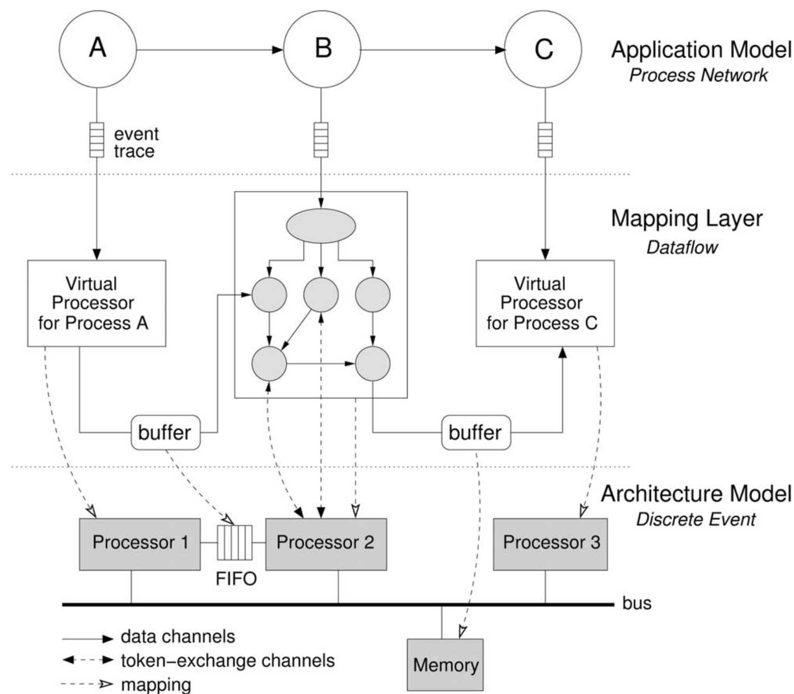


Figure 3.18 *Sesame model layers [42].*

Sesame [42] is a design framework supporting high level modeling and simulation, performance evaluation, and automatic design space exploration. Sesame maintains there model layers as shown in Figure 3.18, including:

- Application model: is specified in PPN with concurrent processes communicating through FIFO channels. The leaf computation code of the process is implemented in C/C++ language while the control codes (loop, data flow control) and interfaces as well as the structure of the network are described by XML-based Y-chart Modeling Language (YML)[15]. An example of a process can be viewed in Figure 3.19(a). Each process contains a list of legal mapping PEs and a computation requirement while each channel contains a communication requirement. These requirements are demanding workloads of computation and communication to mapped components. The workload is presented by traces of application events, which are generated when executing a process to posteriorly drive the associated mapped component in an architecture model. The events include communication events *read* and *write* and computation events *execute*.

- Architecture model: describes a general structure of a platform including allocated PEs and their connections. Components in the architecture model are coded in Perl or SystemC with support of add-on library SCPEx (SystemC Pearl Extension), while the structure is described in YML. The architecture model is used to simulate performance of an application based on events issued by the application model. A description of an architecture structure in YML is illustrated in Figure 3.19(b).

- Mapping layer: is an intermediate layer, which supports the mapping mechanism between application model and architecture model. The layer contains virtual processors and FIFO channels, which are mapped to application processes and channels, respectively. The mapping is unique and can be automatically created from the application model. The mapping layer helps schedule and forward various computation and communication events from the application model to the architecture model in a way that can prevent deadlock happen. The relationship of components between the mapping layer and the architecture model is changeable depending on design purpose. The mapping layer evolves along with the refinement of the architecture model to model more grained events for accurate reflection of the detailed platform implementation, while the application model remains unchanged.

With these modeling layers, Sesame can perform high-level and mixed-level simulation for early performance evaluation. Besides, automatic design space exploration can be implemented with enhanced Strength Pareto Evolutionary Algorithm (SPEA2) [58] to find the Pareto-optimal solutions corresponding to the best mapping between application and platform in terms of performance, power, and cost.

ESPAM [50] is a high-level SW/HW synthesis tool, which handles detailed implementation of a design. The input of ESPAM is platform specification including the general structure of the platform (allocated PEs and connections between them), the process network application model, and the mapping description between them.

In HW part, ESPAM elaborates and implements the communication channels and interfaces of the platform. ESPAM currently support only pre-designed programmable processors, and only automatically generates custom HW descriptions for interfaces and supporting components of the communication network. ESPAM proposes its own set of components including Communication Memory(CM), Communication Controllers(CC) for implementing communication structure compatible with a process network application model. CM is assigned to each processor in the platform, and is designed as a FIFO buffer memory, which only its own processor can write to

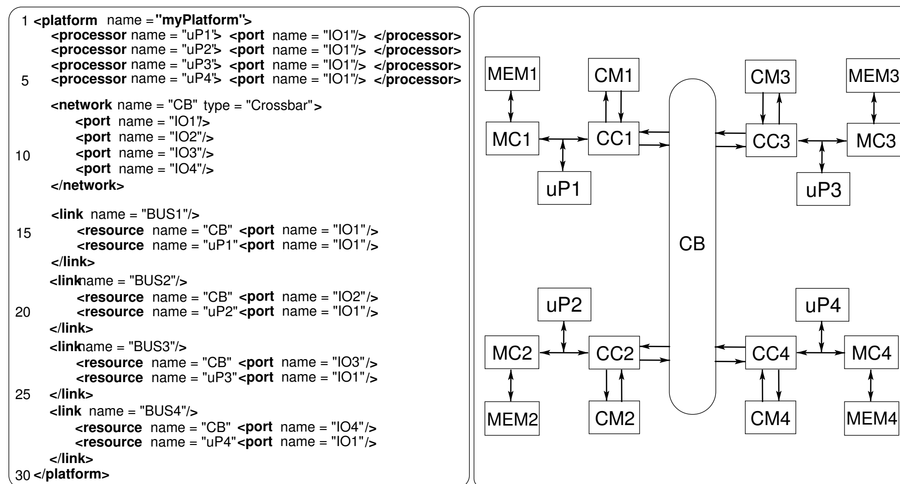
```
1 <process  name ="B"  >
      <port  name = "p1"  direction = "out" />
          <var  name = "out_0" type = "myType"/>
      </port
5     <port  name = "p2"  direction = "in"  />
          <var  name = "in_0" type = "myType"  />
      </port
      <process_code    name = "compute" >
          <arg   name = "in_0"  type = "input" />
10        <arg   name = "out_0" type = "output" />
          <loop   index = "k"  parameter = "N" >
             <loop_bounds  matrix = "[1,  1,0,–2;"
                                     1,–1,2,–1]"/>
             <par_bounds  matrix = "[1,0,–1,384;"
15                                   1,0,  1, –3]"/>
          </loop
      </process_code
   </process  >
                ...
20 <channel   name = CH2 >
      <fromPort name = "p1"/>
      <fromProcess name = "A" />
      <toPort  name = "p2" />
      <toProcess name = "B" />
25 </channel
```

```
1  void  main() {
       for  ( int k=2; k<=2*N–1; k++ ){
           read( p2, in_0, sizeof(myType) );
           compute( in_0, out_0 );
5          write( p1, out_0, sizeof(myType) );
       }
   }

   void  read( byte *port, void *data, int length )  {
10     int  *isEmpty = port + 1;
       for ( int i=o; i<length; i++ ){
           // reading is blocked if a FIFO is empty
           while( *isEmpty ){  }
           (byte* data)[i] = *port; // read data from a FIFO
15     }
   }

   void  write( byte *port, void *data, int length )  {
       int  *isFull = port + 1;
20     for ( int i=o; i<length; i++ ){
           // writing is blocked if a FIFO is full
           while( *isFull ) {  }
           *port = (byte* data)[i]; // write data to a FIFO
       }
25 }
```

(a) Application description and its implementation



(b) Platform description and its implementation

Figure 3.19 *ESPAM HW and SW implementation [50].*

and other processors can read from. CC acts as an interface between processor bus and the main bus, and controls access to the CM. Processors transfer data using blocking *read*/*write* methods, in which each processor writes to its own CM until the buffer is full and reads from other CMs unless the bus resource is not available or the desired buffer is empty. A main bus can be a shared bus or use ESPAM special component named Crossbar Bus (CB). Crossbar Bus implements fast and simple uni-directional connections between CCs with 32-bit width for one direction and two status signals. CB switches to connect two CCs based on round-robin scheduling. The structure of both CC and CB are illustrated in Figure 3.20. Besides, a point-to-point network can be created with each uni-directional channel is implemented by a CM, as shown in Figure 3.21.

In SW part, ESPAM converts YML description of an application into a C/C++

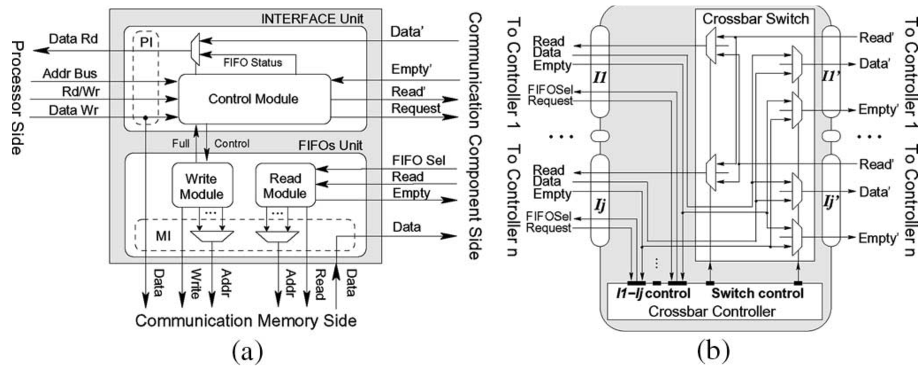


Figure 3.20 *Structure of (a) CC and (b) CB [50].*

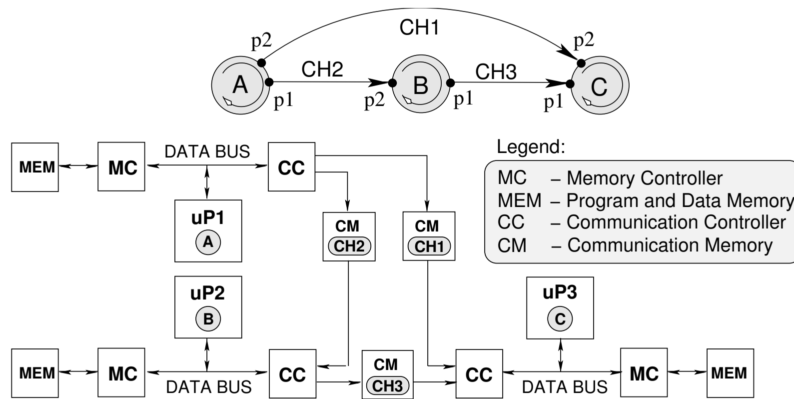


Figure 3.21 *Point-to-Point network implementation [50].*

code plus the blocking *read/write* function implementation and memory map of the platform. For the case when multiple processes are mapped to a single processor, ESPAM statically schedules the processes in the generated C/C++ code instead of using RTOS. The implementations of HW and SW in ESPAM are illustrated in Figure 3.19(b).

The output of ESPAM contains elaborated description of platform topology including detailed communication implementation, the RTL-level description of HW components and program codes for every processors in the platform. The ESPAM output can be further imported in other tools for physical implementation. Currently, ESPAM supports Xilinx Platform Studio (XPS) for FPGA prototyping.

The Motion JPEG (MJPEG) is implemented as a study case for Daedalus [40, 42, 50, 52, 57]. As shown in Figure 3.22, the application model including 8 processes and 10 channels is automaticall generated from the modified sequential code by PNgen. The multiprocessor platform is modeled as up to maximum 4 processors, MicroBlaze and PowerPC tyype combination, crossbar-based communication and distributed memory. Design space exploration process in Sesame results in 10,148 design points, and 11 which have the best performance in execution of 8 128x128

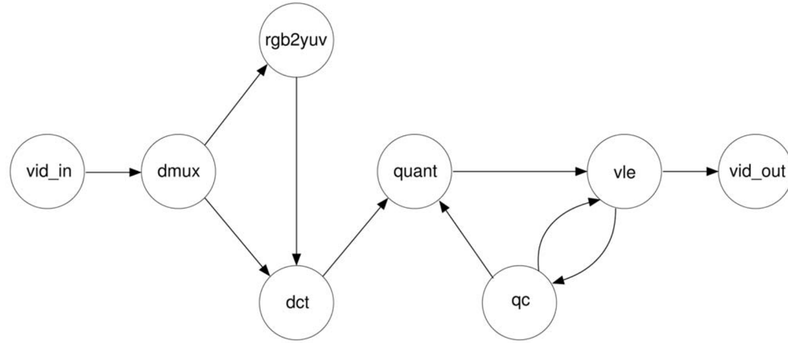
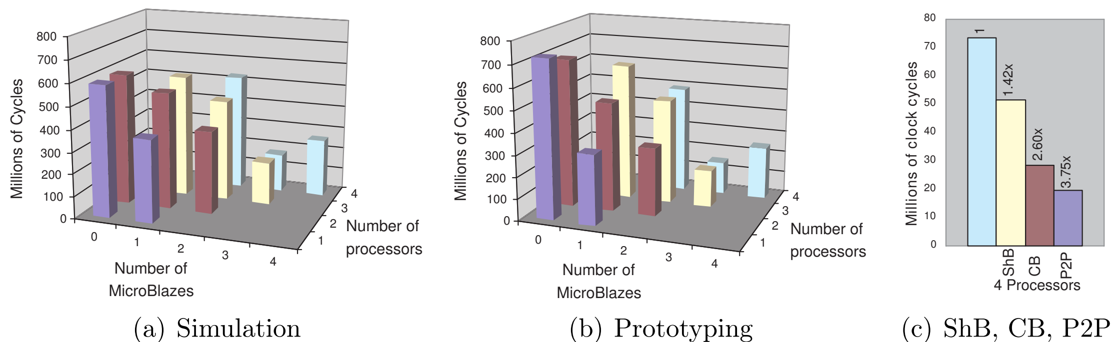


Figure 3.22 *Procees network model of Motion JPEG [42].*

frame sequences are chosen for further implementation. The comparison between the high-level simulation and real implementation on Xilinx Virtex-II FPGA board is depicted in Figure 3.23(a) and Figure 3.23(b), which shows that the error of the simulation is around 13% on average and 28% at the worst. Besides, two other communication types, namely P2P and shared bus, are implemented for 4 MicroBlaze processors platform. Their performances are compared with the Crossbar-based platform with single MicroBlaze platform as a reference. Figure3.23(c) shows that ShB is only 1.42 time faster that the reference, while CB and P2P can achieve up to 2.6 and 3.75 time faster, respectively. The resource utilization of 4 MicroBlaze processors platform with different communication types is also listed in Table 3.6, which shows that the whole system uses around 40% FPGA slices, and efficient proposed communication method with CM and CC of ESPAM occupied only about only 5% in both ShB and CB design. The total development time is written in Table 3.7. The most consuming time task of Daedalus is the exhausting design space exploration in Sesame with 1 hour and 26 minutes, while PNgen takes only 22 seconds for generating process network and ESPAM spends about 25 minutes to synthesis the 11 systems.



(a) Simulation (b) Prototyping (c) ShB, CB, P2P

Figure 3.23 *Daedalus result comparison [52, 50].*

Table 3.6 *Resource utilization of design using 4 MicroBlaze processors [40].*

	Slices	4-input LUT	Flip-Flops	BRAMS
4 Proc. ShB	3640 (39%)	4722 (25%)	2354 (12%)	85 (60%)
4 Proc. CB	3653 (39%)	4748 (25%)	2357 (12%)	85 (60%)
4 Proc. P2P	3263 (39%)	3929 (21%)	2405 (12%)	88 (62%)
4 CCs	288 (2%)	468 (2%)	116 (1%)	-
4 port CB	397 (3%)	587 (3%)	56 (1%)	-
4 port ShB	366 (3%)	541 (3%)	47 (1%)	-

Table 3.7 *Processing Time (hh:mm:ss) [40].*

Tool	PN Derivation	Syst.level DSE	RTL conv	Physical Impl.
PNgen	00:00:22	-	-	-
Sesame	-	01:26:00	-	-
ESPAM	-	-	00:25:00	-
Xilinx Platform Studio	-	-	-	18:29:00

3.5 SystemCoDesigner

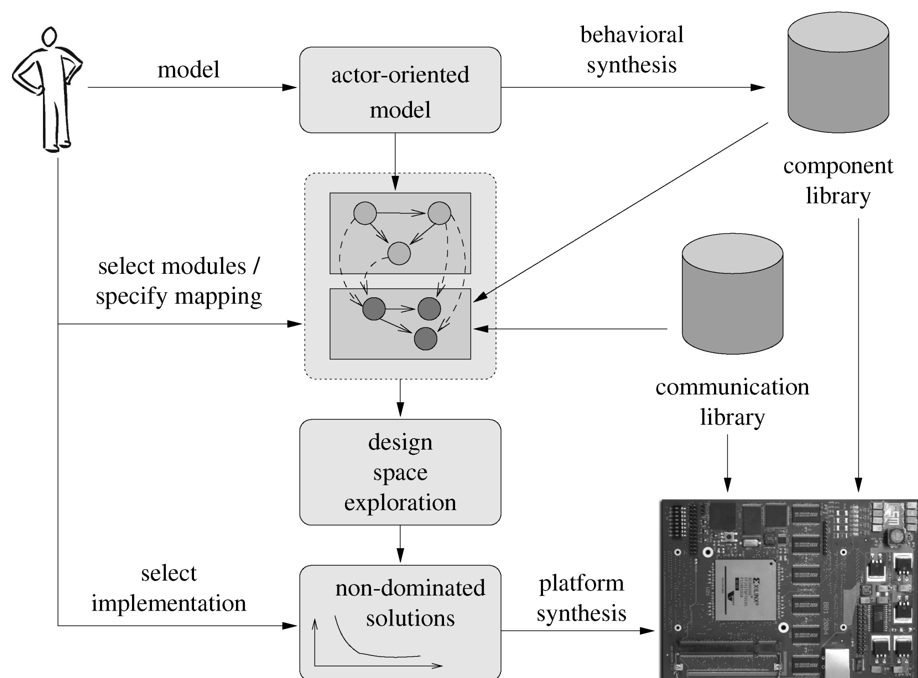


Figure 3.24 *SystemCoDesigner flow [36].*

SystemCoDesigner is a system level design environment developed at the University of Erlangen-Nuremberg, German. The environment supports high-level system modeling and simulation, automatic design space exploration with multiple objectives,

and SW and HW synthesis from abstract model to final implementation.

The design flow of the SystemCoDesigner is illustrated in Figure 3.24. The application input of the environment is described in an actor-oriented model using a special library of SystemC language called SysteMoC [24]. In this model, each concurrent process is an actor consuming, transforming and producing data tokens, which are transmitted via extended FIFO channels named SysteMoC FIFO. An actor in SysteMoC includes three main parts, namely *port*, *functionality* and *communication behavior*. An actor port can be input or output and each is uniquely connected to only one channel, a communication behavior is specified in a FSM called communication state machine, and a functionality is a collection of functions activated on the transition of the communication state machine. The functions are further divided into an *action* part, which performs data consuming, transforming and producing, and a *guard* part, which together with predicate of number of required input token and output space form the boolean expression to activate the corresponding communication state transition. An example of a simple process model in SysteMoC shown in Figure 3.25 includes communication behavior with 2 states, namely *start* and *write*, the transition activation boolean concerning the number of input token $i_2(2)$ and $i_1(1)$, and *guard* $g_{LastPixel}$, and two *actions* $f_{newFrame}$ and $f_{processPixel}$.

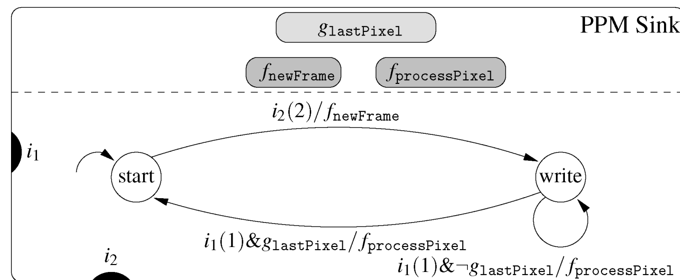


Figure 3.25 *Actor-oriented model [36].*

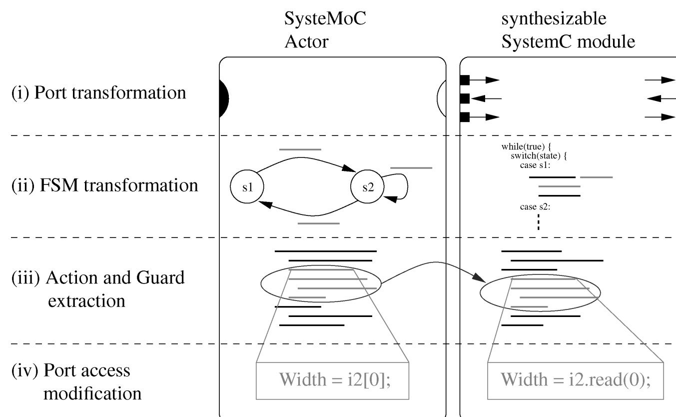


Figure 3.26 *Actor HW implementation [36].*

The actors in SysteMoC model are then synthesized to SW and HW implementations, which are stored in a component library for later use in design space exploration and platform synthesis. The SW is generated by simply converting SysteMoC code into C++ language [31], while in the HW, SysteMoC actor models are first transformed into the synthesizable SystemC modules, in which SysteMoC actor ports are replaced by SystemC signal ports, a communication FSM is converted into an equivalent code, a functionality is transformed into a SystemC module, and port accesses are replaced by corresponding function calls, as shown in Figure 3.26. The SystemC modules are subsequently processed by high-level synthesis tool Forte Design Systems Cynthesizer and Synplify Pro to create RTL descriptions and gate-level netlists, respectively. These actor implementations also contain performance parameters including execution time and cost (areas).

For automatic design space exploration, SystemCoDesigner provides an architecture template for specifying all possible computation and communication components used for the desired platform. The computation components can be the HW implementations of actors or programmable processors from the component library, while communications are selected from another database called a communication library. The architecture template can be generated automatically with one HW implementation for each actor and a selected number of chosen processors and their corresponding communication components. Later changes and modifications can be carried out by designers. Designers can also create mapping constrains, which specify legal mapping lists between actors and channels to computation and communication components. SystemCoDesigner performs the design space exploration process using the multiobjective evolutionary algorithms (MOEA) [49], which results in an approximation of a set of Pareto-optimal design point in term of latency, throughput, and HW area. The HW area of a particular platform is an accumulation of allocated component's, while the timing metric is determined by high-level simulation using Virtual Processing Components (VPC) framework [51], which need to compile only one time for a particular architecture template, and then different allocations and mappings can be configured and loaded at run time, which results in fast simulation for large design points.

The chosen platform from design space exploration is synthesized down to final implementation. SystemCoDesigner currently supports MicroBlaze core for a programmable processor, while the HW modules can be instantiated from actor implementations or pre-designed IPs from the component library. Communication between HW modules is implemented using FIFO primitives, which can be BRAM or LUT based, while fast simplex links (FSLs) is deployed for MicroBlaze cores. A special bridge is used to connect these two buses. Local data transferring in

a MicroBlaze is implemented by read and write methods to local memory buffer. Round-robin policy is used to schedule multiple actors mapped to one processor. The physical implementation can be carried out by Xilinx Embedded Development Kit.

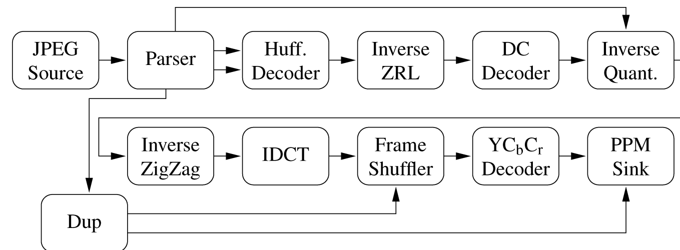


Figure 3.27 *Motion-JPEG block diagram [36].*

The Motion-JPEG decoder has been implemented as a case study for SystemCoDesigner [32, 36]. A block diagram shown in Figure 3.27 includes 11 main tasks, namely Parser, Huffman Decoder, Inverse ZRL, DC Decoder, Inverse Quantization, Inverse Zigzag, IDCT, Dup, Frame Shuffler, YC_bC_r Decoder, and PPM sink. The JPEG stream is first analyzed by the Parser block to extract necessary information, then it is passed through Huffman Decoder, Inverse ZRL, DC Decoder, Inverse Quantization for entropy decoding. Then the data is decompressed by Inverse Quantization, Inverse Zigzag, IDCT. After that, the pixel blocks is re-arranged into raster scan order by Frame Shuffler, and converted into RGB format by YC_bC_r. The final result in Portable Pixmap File is generated by PPM sink.

The chosen architecture template includes 19 HW implementations for every actor, one MicroBlaze processor, BRAM-based and LUT-based FIFO, FSLs bus and corresponding interfaces. As shown in Table 3.8, the design space exploration process runs for 2 days, 17 hours and 46 minutes, and the average simulation time for executing Motion-JPEG streams with 4 176x144 pixels QCIF frames is around 30 seconds. The results are 366 optimal design points. The simulation and implementation results of several designs are shown in Table 3.9. The differences in latency and throughput are claimed to be caused by the schedule overhead and zero-time simulation of guard evaluation of VPC, while the differences in used area is due to post-synthesis optimization and configuration of MicroBlaze processors of a Xilinx tool. However, the results still show acceptable agreement between the SystemCoDesigner simulation and the actual implementation.

Table 3.8 *Design space exploration [32].*

Parameter	Value
Evaluated solution	7,600
Optimal solution	366
Total running time	2d17h46min
Simulation time	30.44s/solution

Table 3.9 *Design space exploration [32].*

Num.of SW actor	0		1		8		all	
	Sim.	Imp.	Sim.	Imp.	Sim.	Imp.	Sim.	Imp.
Latency(ms)	12.61	15.63	25.06	23.49	4,465	6,274	8,076	10,030
Throughput(fps)	81.1	65	40.3	43	0.22	0.16	0.13	0.10
LUTs	44,878	40,467	41,585	35,033	17,381	15,064	2,213	1,893
FFs	15,078	14,508	12,393	11,622	8,148	7,540	1,395	1,086
BRAM/MUL	72	47	96	72	63	63	29	29

3.6 No-Instruction-Set Computer

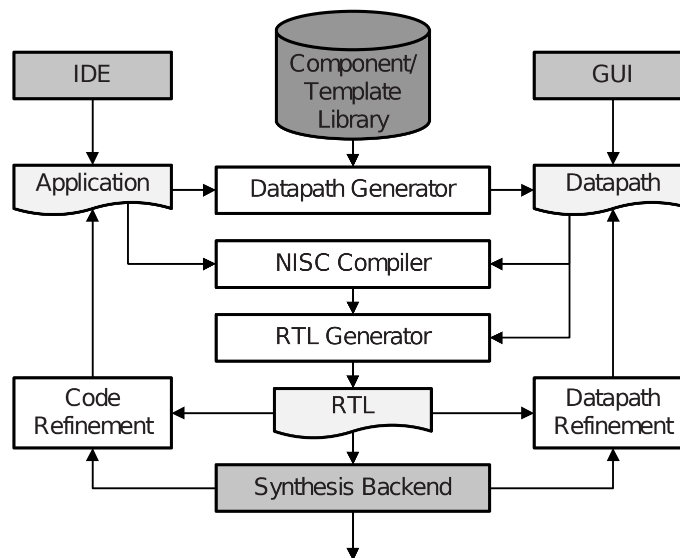


Figure 3.28 *NISC design flow [25].*

No-Instruction-Set-Computer (NISC) is a special HLS tool-set developed at the Center for Embedded Computer Systems, University of California, Irvine. The tool-set implements transformation from a C application to an RTL description of a custom HW with a special NISC datapath architecture. The tool-set also provides simulation and debugger supporting for optimization and refinement.

NISC proposes a special datapath architecture, in which instead of instructions, components in a datapath are directly controlled by signals called control words (CWs) generated by a controller every clock cycle. The controller is usually fixed and composed of Program Counter (PC) register, an Address Generator (AG), and a Control Memory (CMem) storing the CWs as illustrated in Figure 3.29. A typical CW, as shown in Figure 3.30, contains control signal values for each component in the datapath in the fields, and the value can be '0', '1', or 'x', which means 'don't care' or 'idle'. CW can also have several constant fields carrying constant data values from the program. The NISC datapath architecture doesn't require a complex controller with decoder stage and HW scheduler, therefore offers potentially better performance, lower area and power consumption than the conventional ones.

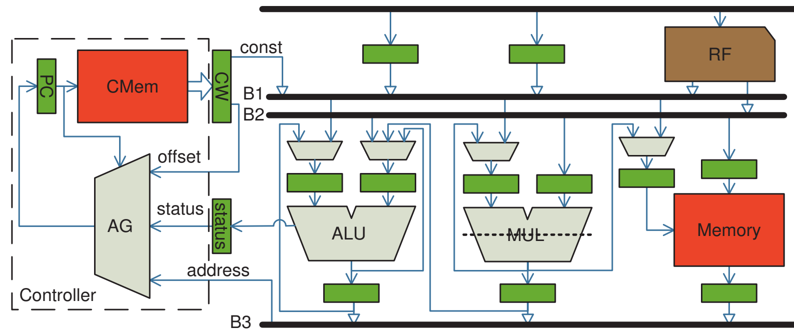


Figure 3.29 *NISC architecture [30].*

RF_ra0	RF_ra1	RF_wa	RF_we	ALU_op	Mux0_sel	Mux1_sel	R0_load	…	Constant

Figure 3.30 *NISC control word [30].*

The NISC design flow shown in Figure 3.28 is composed of three main steps, namely datapath generation, NISC compilation and RTL generation. Currently, the NISC tool-set supports an input application in C language with several restricted features [55] including function pointers, global pointer initialization, standard libraries. Besides, available components in NISC library don't support double and long types, and special operations such as floating point operations, and division must be explicitly supported by the real allocated HW in the datapath.

The NISC datapath can be automatically generated from the application [53], or manually created via a GUI, or reused from a library. For the automatic datapath generation, an application is first pre-scheduled in As Late As Possible form from which statistics of every operation (a number of occurrences in each cycle, data dependency) are extracted to create an initial requirement. This requirement specifies a datapath architecture, which supports maximum parallelism and is composed of components, which can implement the most operation types. NISC tool-set

then performs iteration of evaluation, optimization and refinement (allocated component reduction) of the generated architecture until the specified performance and utilization constraints are satisfied. A manually created or pre-designed datapath architecture can also be automatically optimized based on specific application to meet chosen constraints [27]. An architecture structure is described in XML-based Generic Netlist Representation (GNR) [29, 28]. A component in GNR is represented by component's type, ports, contained components, and 3 aspects specifying behaviors for compiler, simulator and synthesizer. The type can be a basic component such as register, register-file, bus, mux, tri-state buffer, functional-unit, memory-proxy, controller, or a container such as module, NiscArchitecture (top container). Ports are classified as data ports, clock ports, and control ports. A compilation aspect describes the relation between component functions and C program, while the simulation and synthesis aspect describes the simulatable and synthesizable HDL description of the component, respectively. An example of GNR is shown in Figure 3.31.

Given the datapath architecture, NISC cycle-accurate compiler will transform basic blocks of the application into CWs [45]. A basic block is a sequence of operations always executed together, and the compiler will statistically schedule and map operations of a basic block to a component in the datapath to create series of CWs. As mentioned above, using CWs helps reduce controller complexity and enhance system performance, but it also results in an enormous code size. NISC solves this issue using dictionary compression technique [30, 26], in which only a set of unique CW patterns is stored, and a controller addresses them via the small code lookup table (codeLUT), as illustrated in Figure 3.32. Two or more dictionaries can be used for further code size reduction and faster addressing. The extra addressing process requires only one more clock cycle, which has no much effect on overall performance.

Finally, NISC RTL generator will automatically create the RTL description for clocks, connections between modules, and connections between modules and a controller from GNR, and combine them with the basic component descriptions and CWs from the compiler to form the simulatable and synthesizable codes for verification and implementation.

Several experiments have been implemented to evaluate the different features of NISC. Multiple applications including Dijkstra, sha, adpcm_coder, adpcm_decoder, CRC32 and fixed-point MP3 decoders are implemented in Xilinx MicroBlaze and simple NISC architecture (with highest optimization) in Figure 3.28, which run at approximately 100MHz and utilize almost same areas (Table 3.10) [30]. Table 3.11 shows the significant enhancement in NISC performance in comparison with Mi-

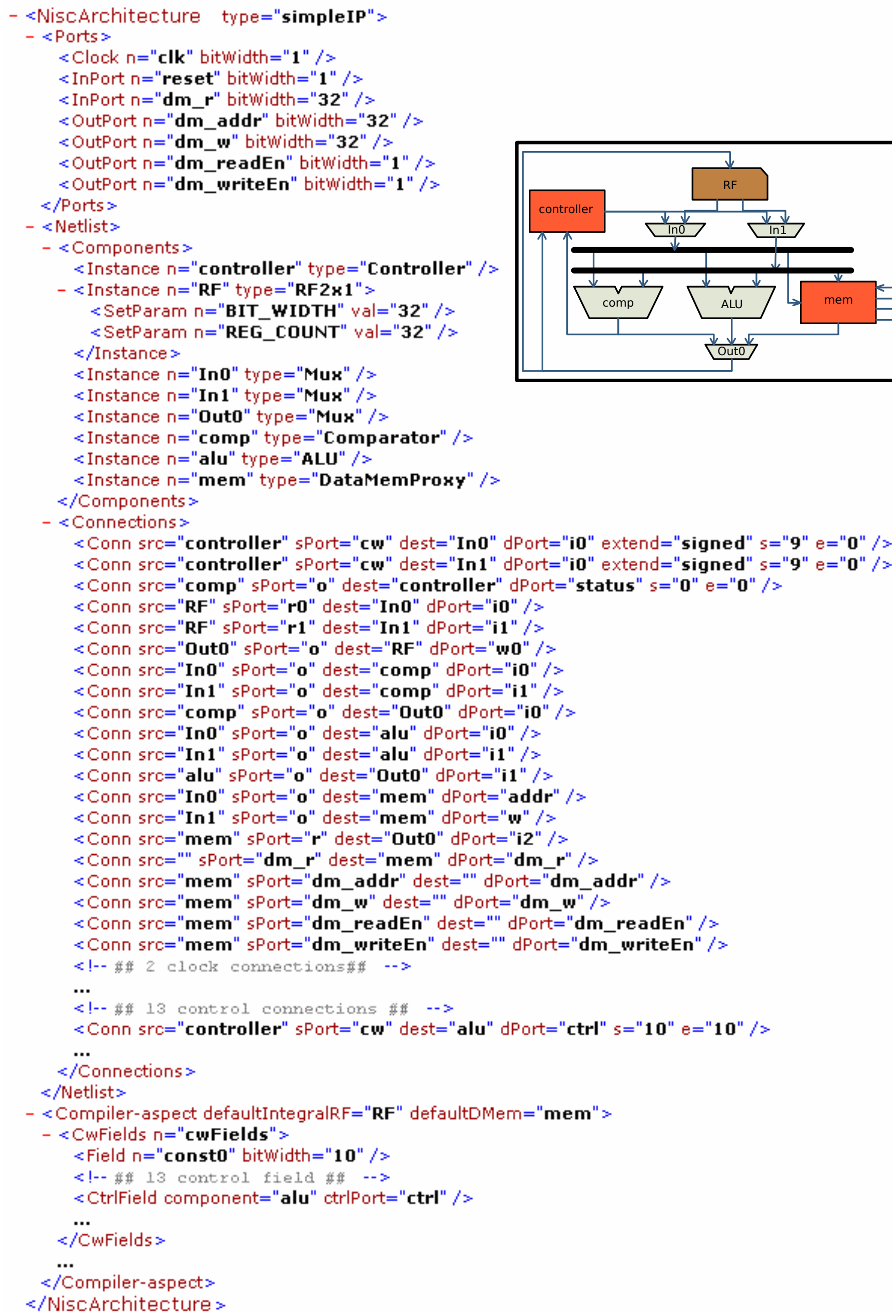
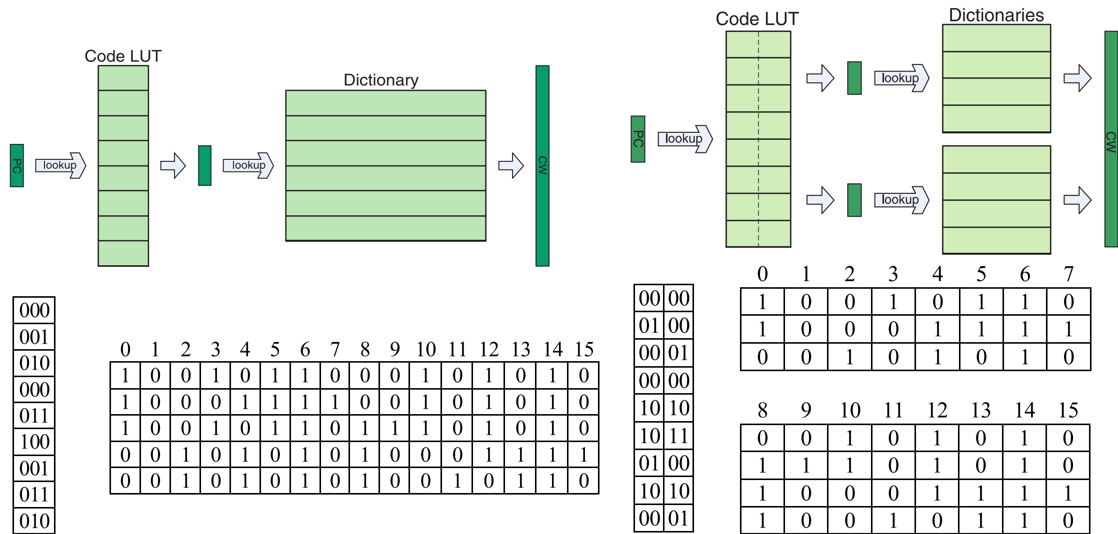


***Figure* 3.31** *IP block diagram and its GNR description [29].*

croBlaze, 5.54 times faster on average. However, The NISC without compression has on average 4 times larger code size as well as occupies more BRAM than MicroBlaze. NISC with one dictionary compression still maintains good performance while significantly reduces the code size, only around 1.16 times larger than MicroBlaze on average. Besides, the MP3 decoder is also implemented on NISCbased MIPS processor (NMPIS) and a NISC customized for the application (NMP3) [29]. The result in Table 3.12 shows dominance of NISC-based processors over MicroBlaze, especially the specifically customized. It also shows the relative relation in size

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	1	0	1	1	0	0	0	1	0	1	0	1	0
1	0	0	0	1	1	1	1	0	0	1	0	1	0	1	0
1	0	0	1	0	1	1	0	1	1	1	0	1	0	1	0
1	0	0	1	0	1	1	0	0	0	1	0	1	0	1	0
0	0	1	0	1	0	1	0	1	0	0	0	1	1	1	1
0	0	1	0	1	0	1	0	1	0	0	1	0	1	1	0
1	0	0	0	1	1	1	1	0	0	1	0	1	0	1	0
0	0	1	0	1	0	1	0	1	0	0	0	1	1	1	1
1	0	0	1	0	1	1	0	1	1	1	0	1	0	1	0

(a) Original CWs



(b) One-dictionary compression (c) Two-dictionary compression

Figure 3.32 *CW with dictionary compression [30].*

between the generated fast simulatable RTL code and the synthesizable one.

Table 3.10 *MicroBlaze and simple NISC [30].*

Processor	Clock Freq. (MHz)	# 4-input LUTs
MicroBlaze	100	1581
NISC	100	1576

Table 3.11 Comparison between MicroBlaze and simple NISC [30].

Application	MicroBlaze		NISC		NISC (Compression)	
	cycles	code size	cycles	code size	cycles	code size
adpcm coder	256748693	1.956KB	74321930	6.960KB	84251684	2.19KB
adpcm decoder	322766405	1.364KB	63082673	5.075KB	66504319	1.59KB
CRC32	209436647	1.264KB	21901993	2.567KB	26008604	0.80KB
dijkstra	25927532	1.928KB	9764682	9.614KB	10631310	2.52KB
sha	183030479	3.156KB	19282976	14.123KB	18371827	4.12KB
Mp3	2668445	44.62KB	897452	216.659KB	927307	63.08KB

Application	MicroBlaze vs. NISC		MicroBlaze vs. NISC Compr.	
	speedup (x)	code size ratio	speedup (x)	code size ratio
adpcm coder	3.45	5.10	3.05	1.12
adpcm decoder	5.12	2.59	4.85	1.17
CRC32	9.56	2.03	8.05	0.63
dijkstra	2.66	4.99	2.44	1.31
sha	9.49	4.47	2.44	1.30
Mp3	2.97	4.86	2.88	1.41v
Average	5.54	4.01	5.21	1.16

Table 3.12 Compasion between MicroBlaze, NISC-based MIPS, and NISC MP3 decoder customized datapath [29].

Processor	Freq (MHz)	Area (%)	cycles (Mil.)	Speedup	Sim. RTL	Syn. RTL
MicroBlaze	100	11	2.7	1	-	-
NMIPS	70	13	0.92	2.04	1981	22300
NMP3	95	17	0.83	3.1	2490	23500

3.7 xPilot

xPilot is a platform-based behavior synthesis framework developed at University of California, Los Angeles. xPilot supports automatic high-level synthesis for customized HWs and mapping for configurable processors and multi-processors with high optimization in performance, power and resource utilization.

xPilot design flow is shown in Figure 3.33. The input of xPilot includes applications describing in synthesizable C or SystemC, and platform descriptions and constraints specifying high-level resources (functional units, connectors, memories) with their delay, latency, area, and power specification. The front-end LLVM compiler firstly converts the application into LLVM intermediate representation, which then is used

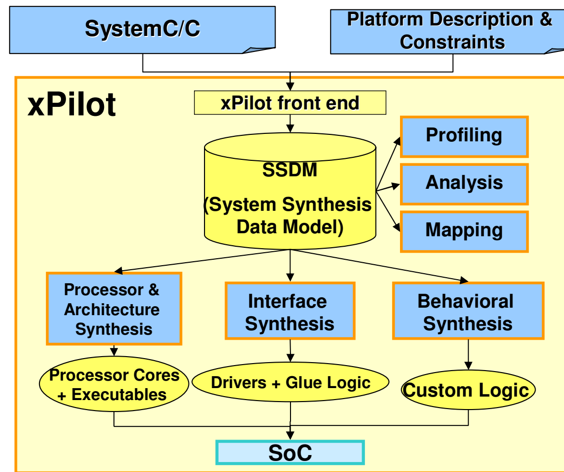


Figure 3.33 xPilot design flow [16].

to construct a xPilot system-level synthesis data model (SSDM). SSDM is a process network, in which concurrent processes are described in a control data flow graph (CDFG) and transfer data through ports and channels. The channels can have several interfaces with various communication protocols. Simulation, profiling and analysing tools can be performed on the SSDM. The HW/SW mapping is done manually by designers.

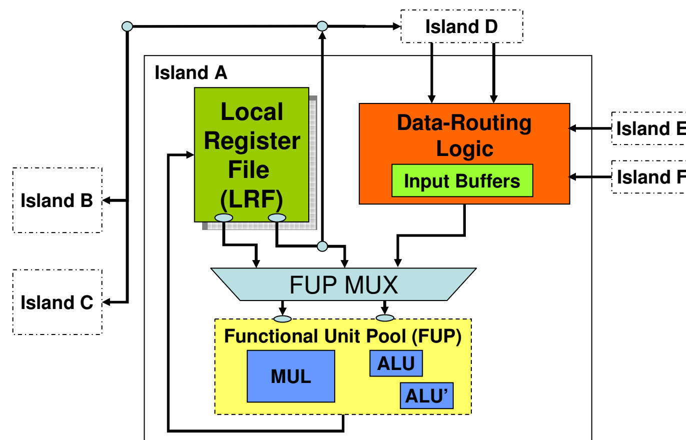


Figure 3.34 Distributed register-file micro-architecture [16].

For behavioural synthesis to customized HW, a sequence of scheduling (assign operations to control states) and binding (assign operations and variables to functional units and registers) is performed, which results in a state transition diagram (STG) and corresponding datapath model. xPilot proposes new system of difference constraints (SDC) based scheduling algorithm [20], which can efficiently optimize a design under various constraints such as resource, frequency, relative timing, longest path latency and overall latency. xPilot also proposes a distributed register-file micro-architecture (DRFM) [16] to reduce the interconnection and multiplexing in

a binding process. DRFM is composed of several so-called islands, each includes a local register file, a functional unit pool FUP and data routing logic, as illustrated in Figure 3.34. The local register file is used to store the local data created by its own FUP, and the data routing logic is used to route data to and from other islands. The DRFM-based binding algorithm tries to assign operations and their associated data differently within one island so that a number of inter-island connections is minimized. Besides, xPilot also employs a mechanism to recognize repetitive patterns in application code and then optimizes the scheduling and binding process bases on this for further resource reduction [19]. The RTL description for implementation and RTL-level SystemC for fast simulation are automatically generated. For configurable processors, the xPilot code generator can exploit the extensive application-specific instruction for performance enhancement [17]. For multiprocessor system, xPilot can perform optimization for latency and resource usage under throughput constraint [18].

Several applications with different characteristics are implemented using xPilot including purely computation DSP kernel PR and MCM, pure control system CACHE, MOTION algorithm in MPEG-1 decoder, IDCT, DWT algorithms in JPEG, and EDGELOOP from H.264 decoder [16]. The designs are implemented on Altera Stratix FPGAs with Quartus II v4.2, and the reports are shown in Table 3.13. The comparisons are also made with another academic high-level synthesis tool named SPARK [11], which shows, on average, about 40% latency improvement and 2 times more area reduction.

Table 3.13 *Implementation reports [16].*

Application	C lines	VHDL lines	LE	Fmax(MHz)
PR	90	600	1349	178.7
MCM	161	1260	2402	152.6
CACHE	295	1277	371	161.6
MOTION	130	1200	888	161.2
IDCT	236	7388	9351	162.9
DWT	180	1371	1862	147.3
EDGELOOP	329	7296	7440	100.1

3.8 GAUT

GAUT is a high-level synthesis tool developed at Universite de Bretagne-Sud, France. GAUT generates RTL description from C application under multiple constraints including data average throughput, system clock frequency, memory architecture and

mapping, I/O timing and FPGA/ASIC target technology. The output of GAUT is a VHDL description for implementation or cycle accurate SystemC for simulation. A special feature of GAUT is the high optimization in the term of resource utilization for multi-mode or multi-configuration application, which changes behaviors or characteristics during run-time. The different behaviors of the application are described by time mutually exclusive tasks explicitly specified by switch-case statements. These tasks can have different throughput constraints. The GAUT joint scheduling and binding will try to maximize and combine the similarities in control logic, functional units and memories of these tasks to form a multi-mode architecture with single FSM and shared datapath with minimum allocated computation and storage elements [10].

The multi-mode architecture of GAUT is tested with several application types including single application with various configurations, namely FFT with multiple points (64, 32, 16, 8), and FIR band-pass filter with multiple taps (64, 32, 16), low-pass filter with multiple taps (19, 15, 11, 7), two applications in single architecture, namely a combination of 16 points FFT and IFFT, a combination of 8 points FFT and IFFT, a combination of LMS16 and 16-taps FIR, a combination of DCT8x8 and matrix multiplication 8x8, a combination of DCT 8x8 and 64-taps FIR. The results are compared with other approaches, namely Cumulative Architecture (CA) and SPACT-MR [14]. Table 3.14 shows that, on average, GAUT can reduce about 44% and 21% area in comparison to CA and SPACT-MR, respectively.

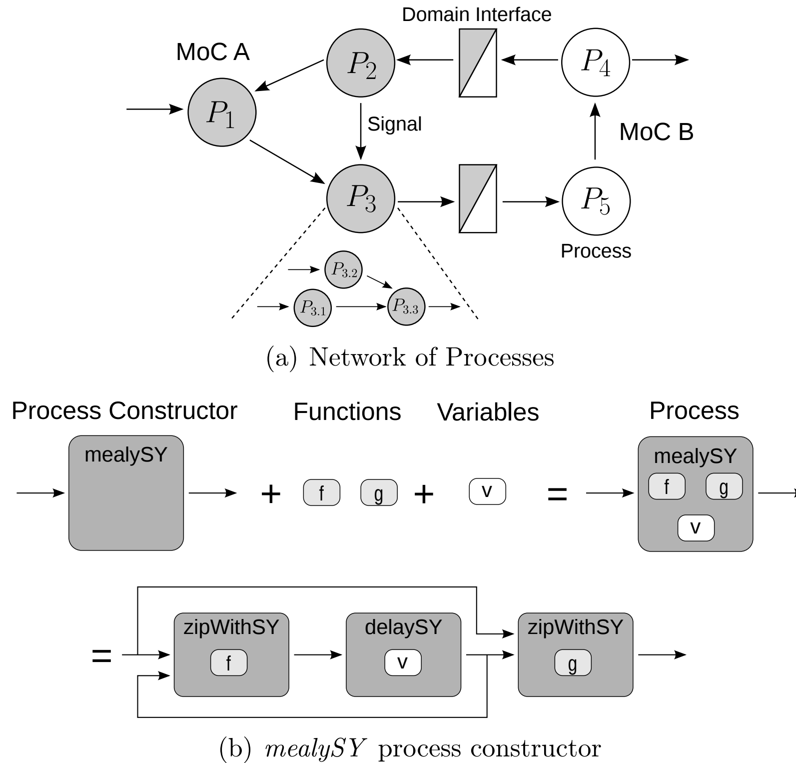
Table 3.14 *Implementation reports [10].*

Application	Area (NAND equivalent)			Reduction (%)	
	CA	SPACT-MR	GAUT	CA	SPACT-MR
FFT(64,32,16,8)	781082	524737	350821	55,1	33,1
FIR(64,32,16)	54132	26634	18786	65,3	29,5
FIR(19,15,11,7)	37701	11103	9249	75,5	16,7
FFT16 + IFFT16	118538	109238	81017	31,7	25,8
FFT8 + IFFT8	36033	31545	25561	29,1	7,4
LMS16 + FIR16	51396	38884	36016	29,9	7,4
DCT 8x8 + FIR64	370115	345813	339881	8,2	1,7

3.9 Formal System Design

Formal System Design (ForSyDe) is a high-level modeling and simulation framework developed at KTH Royal Institute of Technology, Sweden. ForSyDe provides formal, abstract semantics to specify models of a heterogeneous embedded systems in various

MoCs. A ForSyDe specification model can be simulated and exported to other tools for further implementation.



(a) Network of Processes

(b) *mealySY* process constructor

Figure 3.35 *ForSyDe system model [6].*

A system model in ForSyDe shown in Figure 3.35(a) is composed of a structural network of concurrent processes communicating using specifically defined signals. A process can be described by one of four supported MoCs including Untimed/Synchronous Data Flow (SDF), Synchronous (SY), Discrete Event (DE), Continuous Time (CT), which are sufficient for various domain applications, such as data streaming, control-oriented SW, test-bench, or digital and analog components. A process is created only by various abstract blocks named process constructors provided in ForSyDe library. Each process constructor has specific characteristics (number of input/output, HW/SW interpretation) and requires different arguments, which usually are functions and values (initial state, counting number, ...) to create a process, as illustrated in Figure 3.35(b). Basic process constructors, as shown in Figure 3.36(a), include *mapSY* to model combinational process with one input and one output, *zipWithSY* to model multiple inputs combinational process, *delaySY* to model signal delay, *zipSY* and *unzipSY* to combine and separate multiple signals. Other process constructors can be formed by combining basic ones such as *scanlSY* and *scanldSY* to model finite state machines without output decoder, *mooreSY* and *mealySY* to model Moore and Mealy finite state machine. An example *mealySY* constructor is shown in Figure 3.35(b). The signals in ForSyDe are a sequence of

events, each of which is composed of tag and value, as shown in Figure 3.37. A tag is used to specify time or order of events, while a value can be any type, or be marked as absent. Processes with different MoCs are connected by Domain Interface (DI), directly or through multiple layers.

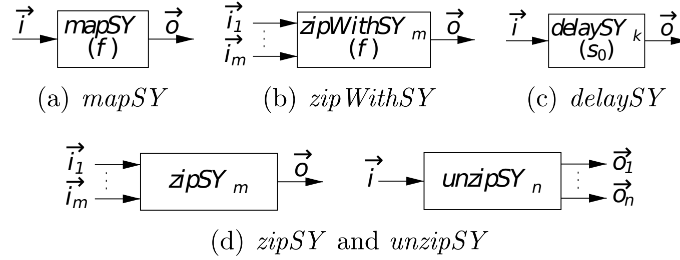


Figure 3.36 *Basic process constructors [47].*

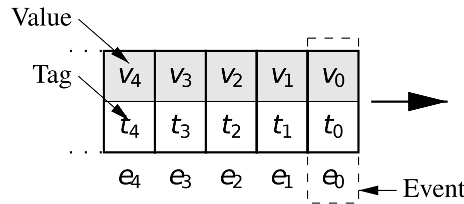


Figure 3.37 *ForSyDe signal [6].*

A model in ForSyDe is formerly implemented by functional language Haskell, but currently has changed to C++-based class library on top of SystemC language, where a process is a module with a single process, a signal is a primitive channel, a process constructor is C++ class template, and so on as shown in Table 3.15. Models in ForSyDe can be simulated based on the Model of Execution for KPNs with blocking writes to bounded FIFOs. Abstract semantics are used to implement computation and communication behaviours of process constructors (and corresponding processes) and DI. A typical implementation of a process contains a *init* stage for memory allocation and variable initialization, iteration stage with *pred* for reading or updating input values, *apply* to transform the values based on defined functions, and *prod* for writing a result to output and synchronizing with system kernel, and *clean* stage for clean up tasks when the process ends. ForSyDe can perform co-simulation with other foreign components modeled in different SWs and languages using a special process constructor named *Wrapper* to wrap the foreign model and implement communication and synchronization between the foreign simulator and ForSyDe simulator [6]. ForSyDe model can be transformed into an intermediate representation in XML and C++ files, which can be used as an input to other design tools for further implementation.

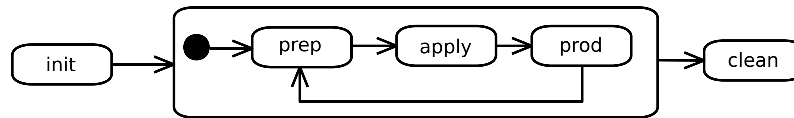


Figure 3.38 *Process implementation [5].*

Table 3.15 *SystemC implementation of ForSyDe [5].*

ForSyDe	SystemC
Process	instantiated module
Signal	channel (*sc fifo*, etc.)
Process constructor	*sc module*-based class
Function argument	*std::function* object
Initial value argument	value passed to class constructor
Polymorphic process and function	template class and template function

3.10 Ptolemy II

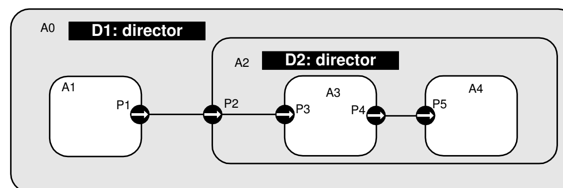


Figure 3.39 *A model in Ptolemy [23].*

Ptolemy II [23] is a modeling framework developed at University of California, Berkeley. Ptolemy II supports modelling and simulating a heterogeneous system composed of different types of MoCs.

A model in Ptolemy II is a hierarchical structure of concurrent basic blocks called actors communicating through port interfaces, as shown in Figure 3.39. An actor can be a leaf node, which actually processes data, or can be a composite one, which contains other actors. Ports can be input, output or both, and each connection between two ports forms a channel. Each input port is assigned a *receiver*, which implements a communication mechanism and can be FIFO queues, mail-boxes, proxies, or rendezvous points. An execution order is only defined locally amongst actors within a composite actor and controlled by a component called *director*. To create heterogeneous models, which can cooperate multiple MoCs, Ptolemy proposes a mechanism, in which within each composite actor there is a homogeneous environment associated with unique MoC called a *domain*, which contains the corresponding director and specifies specific receivers for interior actors. Actors can be reused in various *domains* with different scheduling policies and communication methods. The

currently well-developed domains in Ptolemy II include continuous-time, dynamic dataflow, discrete-event, finite state machines and modal model, process networks with asynchronous message passing, process networks with synchronous message passing, synchronous dataflow, synchronous reactive, and 3-D visualization.

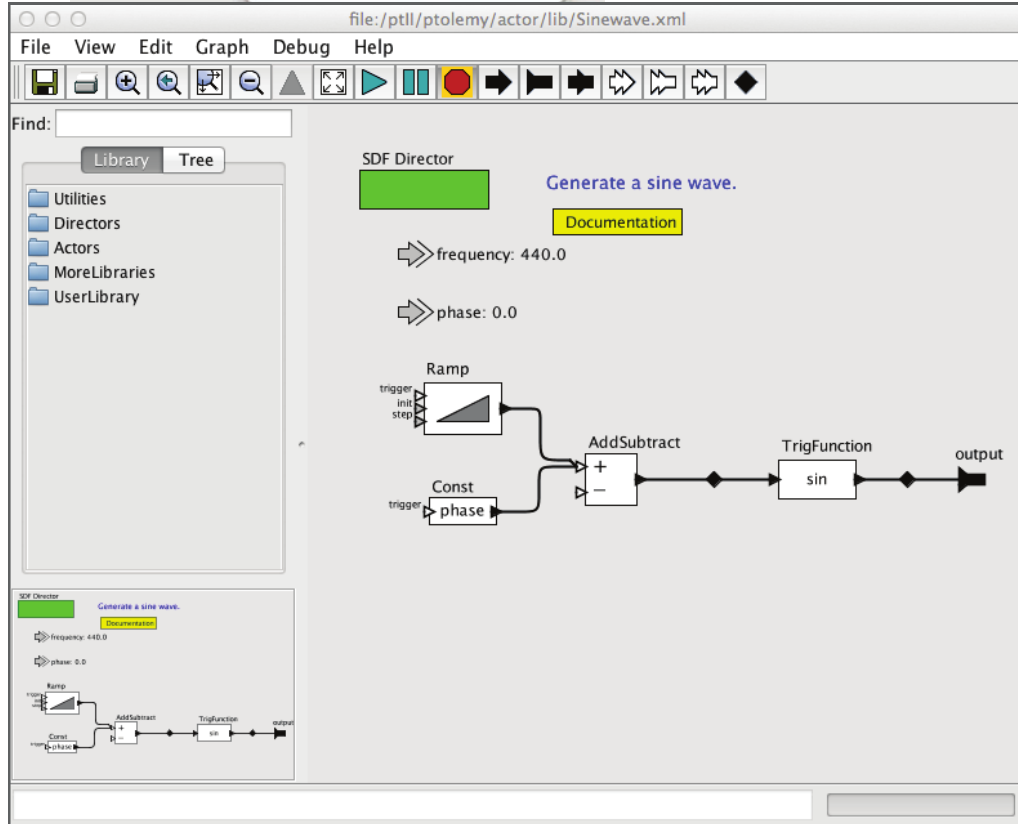


Figure 3.40 *Ptolemy GUI Vergil [43].*

Designs in Ptolemy are carried out in GUI called Vergil, which is depicted in Figure 3.40. Models are formed through drag and drop of various graphical actor blocks. Ptolemy provides several pre-designed actor categories including sources (constant value/string, current time, clock, sine wave, ramp, sequence, pulse ...), sinks (discard, display, plotter ...), and math (add/subtract/multiply/divide, average, max/min, differential, quantizer ...). Complicated mathematical formulas can be conveniently constructed in expression blocks. A model can be stored as an class for later using as instances or modifying to create new subclasses.

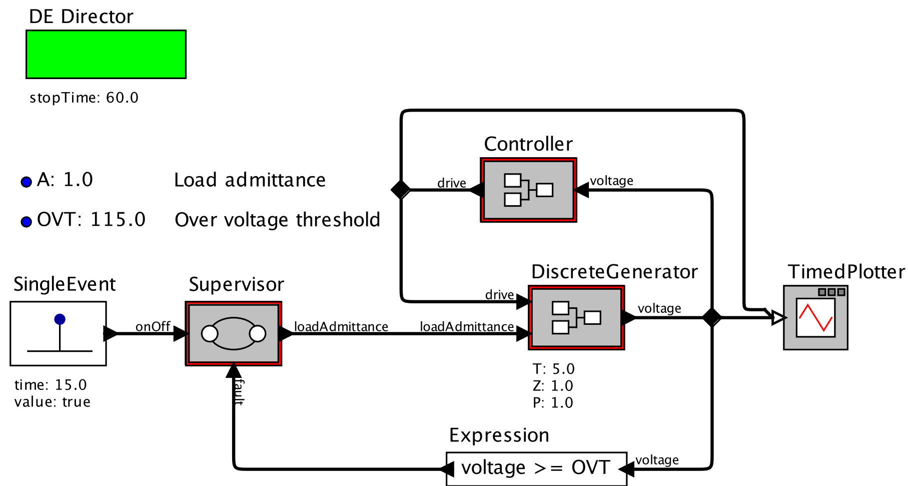


Figure 3.41 The gas-powered generator top model [43].

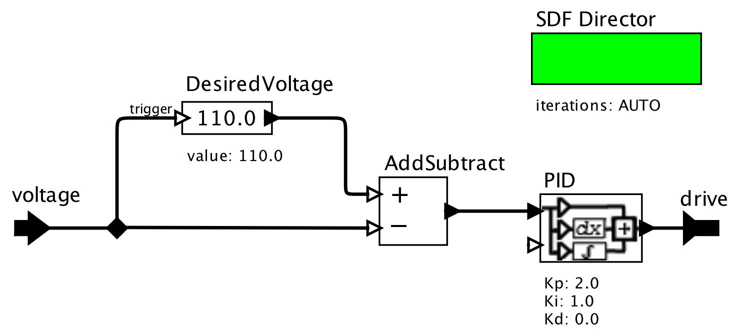


Figure 3.42 The SDF Controller model [43].

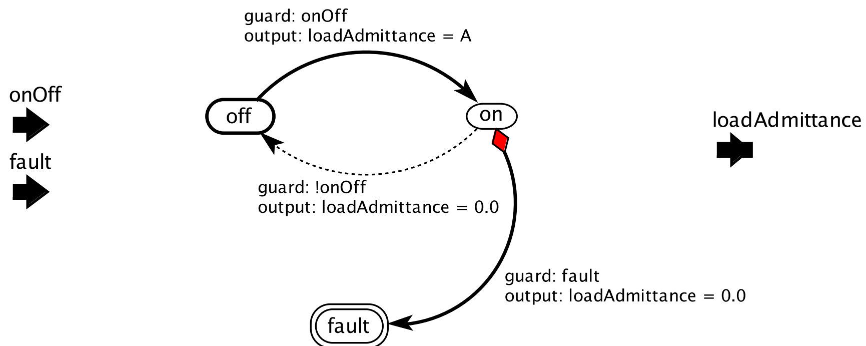
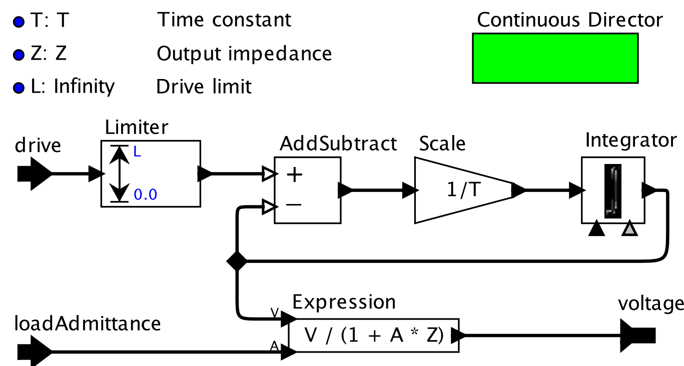


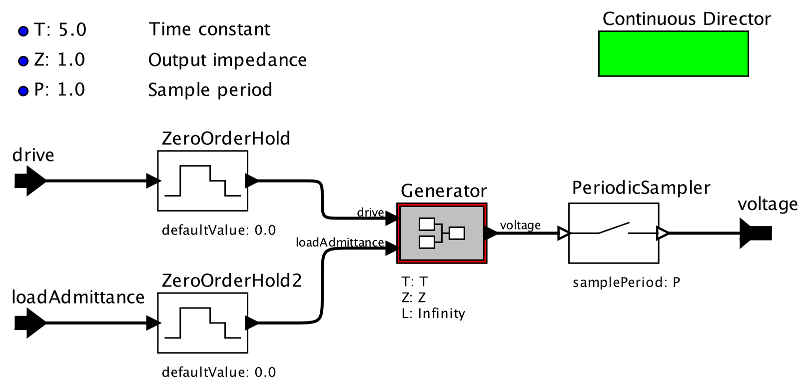
Figure 3.43 The FSM Supervisor model [43].

A simple example of a gas-powered generator system modeled by multiple MoCs is shown in Figure 3.41. A top model is a DE model with 3 main blocks, namely Supervisor, Controller, and Discrete Generator. The Controller drives and regulates the Generator to generate voltage at a fixed level.The Supervisor controls attachment and detachment between a load A and the Generator, and protects the load from over-voltage with a threshold OVT.

The Controller, as shown in Figure 3.42, is a SDF model specifying a PID system, of which input is a difference between a desired voltage 110V and the current Generator's output voltage. The Supervisor, as shown in Figure 3.43, is a FSM model. It is initially in an *off* stage and changes to an *on* stage when there is an *onOff* signal. When in the *on* stage, it observes the *fault* signal, which is activated when the Generator's output voltage is above the OVT, to release the load from the Generator and return to the *off* stage.



(a) The continuous Generator model



(b) The discrete-extented Generator model

Figure 3.44 *The Generator model [43].*

The original Generator, as illustrated in Figure 3.44(a), is a continuous model containing a Limiter followed by a feedback system to increase or decrease a voltage V according to the change of the drive voltage. There is also an expression reflecting the effect of the load A. The Generator is then connected to other blocks, as shown in Figure 3.44(b), so that it can be embedded in the top DE model. These blocks are ZeroOrderHold actors, which hold discrete values constant between consecutive events, and a PeriodicSample actor, which converts continuous values into discrete values.

A plot in Figure 3.45 presents a result of the model. At the beginning, the Controller drives the output voltage to the desired value 110V. After that, the load is connected

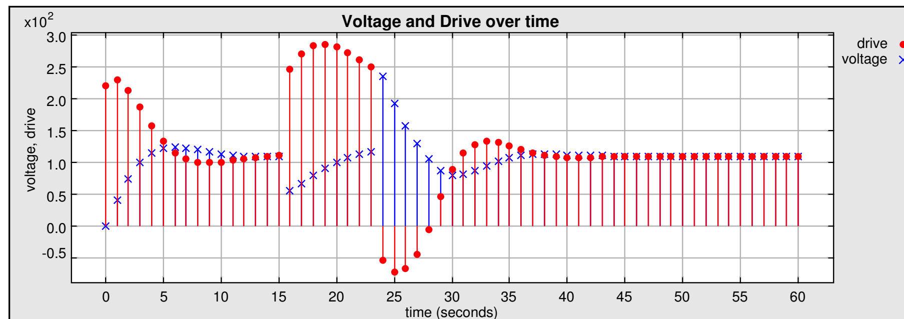


Figure 3.45 *Result of the model [43].*

at time 15, which causes the output voltage to suddenly drop. The drive voltage surges to raise the output voltage back to 110V, which consequently causes voltage-overload at time 24. The Supervisor then disconnects the load, and the Controller regulates the output voltage back to 110V.

4. SYSTEM DESIGN FRAMEWORK COMPARISON

4.1 Comparing metrics

Various metrics are selected for comparison between all the listed tools and frameworks:

- Target domain (Table 4.3): describes an aimed architecture of a tool or framework, which can be a uni-processor system containing single processor core and several peripherals, or a heterogeneous SoC containing different types and numbers of processors and HW IPs along with complicated communication topologies and memory hierarchies.
- Purpose (Table 4.3): describes specific capabilities of a tool or framework within its target domain, which includes modeling, early architecture exploration, simulation and verification, and HW and SW synthesis. Model can be computation architecture, communication architecture, application or design constraint. Early architecture exploration can be automatic or manual. HW and SW synthesis denote the ability to generate from an architecture model down to HW implementation (HDL description of PE, CE, ...) and from an application model down to SW implementation (C/C++ code, binary image, ...).
- Architecture exploration (Table 4.4): describes supporting features of frameworks in the early architecture exploration phase, which includes capabilities to evaluate various component allocation, application mapping, and task scheduling, as well as metrics for performance estimation of the designs.
- Model accuracy (Table 4.4): describes model accuracy supported by a tool or framework, which can be time-estimation or cycle-accurate.
- Model and design language (Table 4.6): describes model types and language used by a tool or framework for system development. Using languages can be different for an application, a HW component, and a platform structure.

4.2 Tools and framework comparison

The summaries of all tools and frameworks are shown in Table 4.1. Most of the projects were started around the early 2000s, and lasting from 5 to 10 years. However, there are a few, which have remained active with recent publications and resource updates, namely Daedalus, ForSyDe and Ptolemy II. The developing languages usually are the common ones like Java, C/C++, and Qt. A GUI is widely supported amongst these tools and frameworks, fully or partially, except a few like Daedalus, Metropolis, which depends entirely on scripts and commands for operation.

These tools and frameworks can be grouped into three categories including design flow tools (SCE, ESE, Metropolis, Daedalus, SCD), which cover most or almost a whole system design steps from modeling, exploration, simulation, verification to implementation, SW and HW synthesis tools (NISC, xPilot, GAUT), which focus on generating specific SW and HW implementation from application programs, and modeling tools (ForSyDe, Ptolemy II), which aim to model, simulate and evaluate applications at the high-level abstraction.

All of the design flow tools support heterogeneous platform development with computation and communication separation in both architecture and application modeling, and design space exploration mechanisms. Noticeably, due to ESPAM component limitation, Daedalus is currently restricted to systems with general-purpose processors only (no customized HW IPs).

Daedalus and SCD, given an initial configuration of available resources and constraints, can automatically evaluate numerous design points regarding component allocation, and task mapping and scheduling to generate the optimal ones in terms of performance, area, or cost metrics, whereas in SCE, ESE and Metropolis, the process must be done manually by designers.

Unlike other tools, in which components can be freely chosen and allocated to form a desired platform, Metropolis design process usually starts with a fixed, predefined architecture template with some configurable parameters. This, on the one hand, restricts a design exploration of the tool to mapping process only, but on the other hand, encourages re-usage of design patterns.

For evaluation of designs, Metropolis and Daedalus cover three metrics including timing, power, and cost/area, SCD supports less with timing and power, while SCE and ESE only provide timing estimation. Only SCE and ESE model system at both TLM and CA abstraction levels, while the remainder support only the former.

SCE and ESE are fully integrated with tools for HW and SW implementation, which can gradually refine systems from high-level abstraction models to cycle-accurate models ready for implementation. In contrast, Daedalus requires all allocated components must be pre-designed and available in a library, and it simply generates trivial glue and interface codes to connect those together. SCD contains only a code generator while leaving the HW synthesis for a third-party software named Forte Cynthesizer. Metropolis itself solely features system modeling, simulation and verification, so it needs to cooperate with other back-end tools for the complete design flow. SCE and ESE often use RTOSes to manage multiple tasks mapping on a single PE, while Daedalus prefers to statically schedule them in generated codes, and SCD uses a simple round-robin mechanism.

Models used amongst these tools and frameworks are varied from process network PN, PPN to dataflow graph or state machine, which are influenced, more or less, by target applications (multimedia streaming, automation, control,...). Most require input applications to be written in a special form (SpecC in SCE, metamodel in Metropolis, SysteMoC in SCD, SANLP C/C++ in Daedalus), which imposes great challenge and difficulty for developing new program as well as re-using the old ones. Moreover, these requirements are not feasible for all applications, which limits the usage of these tools and frameworks. Only ESE accepts an original C/C++ program. SystemC is widely used as the TLM implementation language during the development, while XML or XML-based languages are used for storing platform structure information.

Several study cases ((Table 4.5) show that these design flow tools have greatly reduced the development time, lasting around several hours to a couple days, mainly due to high automation in many design steps and utilization of high-level abstraction models with fast simulation time, and effortless adjustment. Furthermore, the design quality is also guaranteed by accuracy of models, which are shown around 5% to 16% difference compared to real board implementation.

NISC and xPilot support both HW and SW synthesis. However, NISC can only generate code for its own special HW architecture, and SW generation in xPilot is a part of whole processor-based synthesis. NISC, xPilot and GAUT allow to create only a single core system implementation. Each of these tools uses different techniques to improve performance and reduce area utilization, namely no-instruction code and dictionary compression in NISC, distributed register-file micro-architecture in xPilot, and multi-mode architecture in GAUT. The study cases of these tools in Table 4.5 show promising results. Most of them accept C or subsets of C language as inputs.

ForSyDe and Ptolemy II focus on modeling and simulating applications at high-level abstraction. Both allow combination of multiple MoCs in a single system model, but use different mechanisms. ForSyDe uses special components called domain interfaces to implement connections between different MoCs modules, while Ptolemy separates them in hierarchical, composite blocks. ForSyDe supports only four MoCs, namely SDF, SY, DE, and CT, which it believes to be sufficient to model most of application domains, whereas Ptolemy II develops numerous MoCs like PN, DE, SDF, SE, FSM, ... dedicating to its interested fields. A ForSyDe model is built on top of SystemC language, whereas Ptolemy's is described by XML-based form.

For resource availability, there are seven tools and frameworks, namely ESE, Metropolis, Daedalus, SCD, ForSyDe and Ptolemy, publicly publish their software. SCE, xPilot, GAUT require requests to the developing teams for resources, while NISC download link is unavailable. Most of the releases of these tools and frameworks are outdated, back to about five to seven years ago. Only Daedalus, ForSyDe and Ptolemy are active and recently updated. ESE, and SCD provide pre-built binary files, while Metropolis, Daedalus, ForSyDe and Ptolemy come with source files.

Table 4.1 *Framework summaries.*

Name	Developer	Impl.	Features
SCE (1997 - 2003)	UC Irvine	Qt	GUI/Command-line shell Profiling/Estimation tools SpecC compiler and simulator Hardware/Software synthesis Formal Model Algebra based verification tool Website: *ecs.uci.edu/∼cad/sce.html*
ESE (2001 - 2008)	UC Irvine	Qt	GUI/Command-line shell SystemC compiler and simulator Functional/Timed TLM generation tool Hardware/Software synthesis Website: *cecs.uci.edu/∼ese/*
Metropolis (2002 - 2008)	UC Berkeley	Java, C/C++	Metamodel language SystemC simulator Quasi-Static Scheduling HLOC checker & SPIN interface Website: *embedded.eecs.berkeley.edu/metropolis*
Daedalus (2006 -)	UVA, Leiden University	-	Program parallelization Automatic design space exploration Hardware/Software synthesis Website: *daedalus.liacs.nl/*
SCD (2003 - 2013)	University of Erlangen-Nuremberg	-	GUI VPC fast simulation and evaluation Automatic design space exploration Hardware/Software synthesis Website: *informatik.uni-erlangen.de/research/scd*
NISC (2003 - 2008)	UC Irvine	-	GNR structure CW datapath architecture Automatic datapath generation RTL generation Website: *ics.uci.edu/∼nisc/*
xPilot (2003 - 2008)	UC Los Angeles	-	Registerfile microarchitecture (DRFM) DRFM-based & Pattern-based optimization Custom HW synthesis SW synthesis for configurable processors and multi-processors Website: *cadlab.cs.ucla.edu/soc/*
GAUT (1993 - 2007)	UBS	Java	GUI Multi-mode architecture HW synthesis Website: *hls-labsticc.univ-ubs.fr/*
ForSyDe (1999 -)	KTH	C/C++	High-level system modelling and simulator Supported multiple MoCs Website: *forsyde.ict.kth.se/trac*
Ptolemy II (1996 -)	UC Berkeley	Java	GUI High-level system modelling and simulator Supported multiple MoCs Website: *ptolemy.eecs.berkeley.edu/ptolemyII/*

Table 4.2 *Framework resources.*

Name	Resource	Release date	Content
SCE	- [1]	-	-
ESE	ese-2.0	2008	Installation file
Metropolis	metropolis-1.1.3	2008	Source file
Daedalus	PNgen	2012	Source file
	sesame	2014	
	darts	2012	
	Espam	2012	
SCD	scd-0.1	2006	Installation file
NISC	- [2]	2008	-
xPilot	- [1]	-	-
GAUT	- [1]	-	-
ForSyDe	ForSyDe-SystemC-0.4.0	2014	Source file
Ptolemy II	Ptolemy II 10.0.1	2014	Source file

[1] available via request only

[2] empty download link

Table 4.3 *Comparison of purposes and targets.*

	Target		Purpose									
	Uni-processor system	Heterogeneous SoC	Computation Architecture modelling	Communication Architecture modelling	Application modelling	Constraint modelling	Early Architecture exploration	Simulation	Mixed-level simulation	Verification	SW synthesis	HW synthesis
SCE	x	x	x	x	x		M	x		x	x	x
ESE	x	x	x	x	x		M	x			x	x
Metropolis	x	x	x	x	x	x	M	x	x	x		x[1]
Daedalus	x	x	x	x	x	x	A	x	x		x	x
SCD	x	x	x	x	x	x	A	x			x	x[2]
NISC	x					x		x			x	x
xPilot	x	x				x					x	x
GAUT	x					x						x
ForSyDe	x	x			x			x				
Ptolemy II	x	x			x			x		x		

M= Manual; A= Automatic

[1] UCLA's xPilot tool

[2] Forte Cynthesizer

Table 4.4 *Comparison of architecture exploration and model accuracy.*

	Architecture exploration							Model accuracy	
	Application mapping	PE allocation	Task scheduling	Communication scheduling	Communication topology	Memory hierarchy & mapping	Performance estimation	Communication architecture model	Computation architecture model
SCE	x	x	x	x	x	x	T	Te, Ca	Te, Ca
ESE	x	x	x	x	x	x	T	Te, Ca	Te, Ca
Metropolis	x	x	x	x	x	x	T, P, C	Te	Te
Daedalus	x	x	x		x	x	T, P, C	Te	Te
SCD	x	x	x		x		T, C	Te	Te
NISC								Ca	Ca
xPilot								Ca	Ca
GAUT								Ca	Ca
ForSyDe									
Ptolemy II									

T=Timing metric; P= Power metric; C= Cost metric

Te= Time-estimation; Ca= Cycle-accurate

Table 4.5 Comparison of case study.

	Application	Platform	Sim. time	Size	Dev. time	Notes
SCE	JPEG encoder + Vocoder	1 CPU, 1 DSP, 1 DCT HW, DMA	92.3s[1]	21,445[1][2]	8.99s[3][5]	5.15% average performance difference compared to board prototype
ESE	MP3 decoder	1 MB	1s[4]	-	31s	7% average performance difference compared to board prototype
		1 MB, 1 DCT HW	22s[4]	-	50s[4][5]	
		1 MB, 1 DCT HW, 1 IMDCT HW	25s[4]	-	47s[4][5]	
		1 MB, 2 DCT HW, 2 IMDCT HW	36s[4]	-	71s[4][5]	
Metropolis	Motion JPEG	MB + FSL with various task schedulings	165,965[6][7]	6386[7][8]	-	8% average performance difference compared to board prototype
Daedalus	Motion JPEG	Various combinations of MB and PowerPC through design space exploration	-	-	1h51m22s[3]	13% average performance difference compared to board prototype, proposed CM&CC communication use 5% of total area size and enhance speed from 2.6 to 3.75 times
SCD	Motion JPEG	various combinations of MB and custom HWs of main tasks through design space exploration	-	-	2d17h46s[3]	16% average performance difference compared to board prototype

	Application	Platform	Sim. time	Size	Dev. time	Notes
NISC	adpcm coder	NISC architecture with compression	84,251,684 (6)	2.19 (8)	-	NISC architecture with compression technique, on average, is 5.21 times faster and 1.16 times larger than MB
	adpcm decoder		66,504,319 (6)	1.59 (8)	-	
	CRC32		66,504,319 (6)	1.59 (8)	-	
	dijkstra		10,631,310 (6)	2.52 (8)	-	
	sha		18,371,837 (6)	14.12 (8)	-	
	MP3		9,307 (6)	63.08 (8)	-	
xPilot	PR	DRFM-based architecture	-	1349 (9)	-	
	MCM		-	2402 (9)	-	
	CACHE		-	371 (9)	-	
	MOTION		-	888 (9)	-	
	IDCT		-	9351 (9)	-	
	DWT		-	1862 (9)	-	
	EDGELOOP		-	7440 (9)	-	

	Application	Platform	Sim. time	Size	Dev. time	Notes
GAUT	FFT(64,32,16,8)	GAUT multi-mode architecture	-	350821 (10)	-	GAUT, on average, reduces 42% and and 13.5% area compared to CA ans SPACT-MR architectures
	FIR (64,32,16)		-	18786 (10)	-	
	FIR (64,32,16)		-	18786 (10)	-	
	FIR (19,15,11,7)		-	9249 (10)	-	
	FFT16 + IFFT16		-	81017 (10)	-	
	FFT8 + IFFT8		-	25561 (10)	-	
	LMS16 + FIR16		-	36016 (10)	-	
	DCT 8x8 + FIR64		-	339881 (10)	-	
ForSyde	-	-	-	-	-	
Ptolemy II	-	-	-	-	-	

(1) final TLM mode
(2) line of code
(3) total value
(4) TLM model
(5) model generation
(6) cycles
(7) average value
(8) code size in KB
(9) LE
(10) NAND equivalent

Table 4.6 *Comparison of model and design language.*

	MoC	Design Language		
		Application	Component	Platform Structure
SCE	PSM	SpecC	SpecC → Verilog	XML
ESE	PSM	C/C++ → SystemC	XML → Verilog	XML
Metropolis	PN	Metamodel	Metamodel	Metamodel
Daedalus	PPN	C/C++[(1)] → YML	Perl/SystemC + SCPEx → VHDL	YML
SCD	DDF	SysteMoC → SystemC	SystemC/XML	XML
NISC	-	C → CW	RTL	GNR
xPilot	SSDM/STG	SystemC/C	VHDL	-
GAUT	DFG	C	VHDL	-
ForSyDe	Various[(2)]	Haskell/SystemC	-	-
Ptolemy II	Various[(3)]	XML	-	-

PSM=Program State Machine PN=Process Network DDF=Dynamic Dataflow

PPN= Polyhedral Process Network SSDM=System-level Synthesis Data Model

DFG= Dataflow graph STG=State Transition Diagram

[(1)] Constrained in form of Static Affine Nested Loop Programs

[(2)] Synchronous, Untimed/Synchronous Data Flow, Discrete-Event, Continuous Time

[(3)] Continuous Time, Dynamic Dataflow, Discrete-event, Finite State Machines, Process Networks, Synchronous Dataflow, Synchronous Reactive, 3-D Visualization, Continuous Time

5. EXPERIMENTATION

The experimentation is performed with three design flow frameworks, namely ESE, Daedalus, and SystemCoDesigner. These tools are experimented for usage, performance and practicality.

5.1 ESE

The current version of ESE is ESE 2.0, which was released almost 7 years ago, in 2008. The installation of ESE is simply an extraction of pre-built Linux binary files along with several supporting packages. The old provided packages contain a couple bugs and compatible problems. Specifically, ESE 2.0 uses a SystemC 2.2.0 package, which has issues with recent GCC compilers (which has been fixed in later SystemC version), and a *llvm-gcc* compiler package, which is defective and has been deprecated. The tool is pre-built upon these packages, so any attempt to replace them with up-to-date ones will cause a failure. For error-free execution, it is recommended to run the tool in already-tested environments including 32 bit Fedora 3 or RHEL 4.

The released version of ESE is not complete with several features missing. The tool only supports front-end design with platform modeling, automatic generation of functional TLM and timed TLM, and simulation. The core of ESE software includes 4 parts:

- GUI: is shown in Figure 5.1. It provides graphical environment for designers to perform platform development and application mapping, configure parameters, invoke other tools to refine the design, and observe statistic figures. The GUI contains PE Window, which shows various hardware and software component models for platform development, Channel Window, which organizes allocated communication buses and channels between processes in PEs, PE Window, which manages the mapped processes in allocated PEs (source files, process ports, ...), Main Window, which graphically presents the platform, and Menu Bar comprises shortcuts for configurations and other tools.

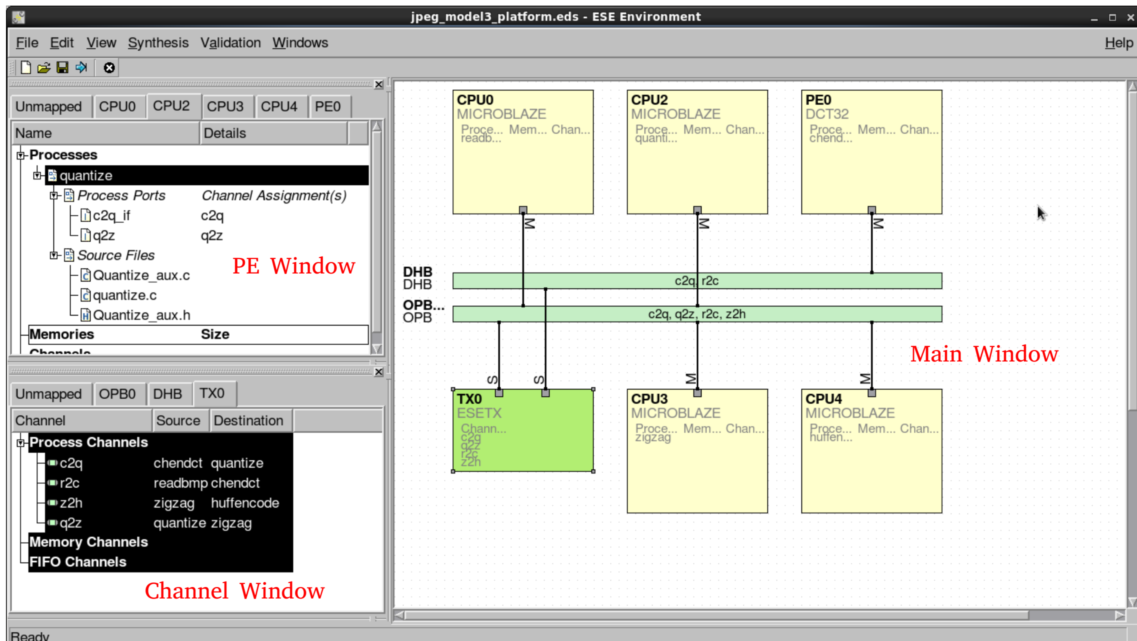


Figure 5.1 ESE GUI.

- Database: the current database appears in the GUI includes Processing (ARM9, Microblaze, pre-designed IMDCT36 and DCT32 IP), Communication(Ethernet, I2C, CAN, DHB, OPB, AMBA, RS232, CYNWP2P, FSL), Memory (Xilinx BRAM, ZBT 512Kx32 static RAM), CE (ESE Transducer,AMBA-APB), Software (Files system, STDIO and Math library, mb-gcc crosscompiler, Microblaze HAL, Xilkernel OS). However, there are only few that has description files for TLM model implementation (Microblaze, IMDCT36, DCT32, Ethernet, I2C, CAN, DHB, OPB, AMBA, RS232, CYNWP2P, FSL, Transducer, Xilkernel OS). None has a description file for physical implementation.

- *tlmgen*: a tool generating functional TLM. It replaces simple API functions in form *send(data ptr, length)*/*recv(data ptr, length)*, which are used to transfer data between concurrent processes in an application program into corresponding code implementations according to mapped channels. It embeds the processes into *sc_modules*, creates code for buses, then combines all into top module, which is ready for compilation and simulation.

- *tlmest*: a tool generating timed-TLM. It first invokes a LLVM compiler to convert the functional TLM into LLVM Intermediate Representation (IR) code (or LLVM operations) in CDFG form. After that, with a performance attribute (in clock cycle) of each LLVM operation and transfer in description files of components, the tool estimates and annotates delays into the code, then converts it back to C program by LLVM code generation and wrapped it with SystemC to create timed TLM.

ESE claims to accept common C/C++ applications with no special requirement. However, there may be hidden compatible necessity for LLVM infrastructure used during the timed-TLM generation. ESE was tested with fairly complex and state-of-the-art video encoder named Kvazaar, which resulted in unknown error during the time estimation phase, probably because of incompatibility.

JPEG encoder application is implemented using ESE 2.0. The general block diagram of JPEG encoder is shown in Figure 5.2. The encoder first reads the BMP image and divides it into block of 8x8 pixels. These blocks then undergo the 2 dimensional Discrete Cosine Transform (DCT) to convert from the spatial domain to the spectral domain where information can be processed for compression. After that, the DCT data is rounded by quantization matrix, and then re-arranged to form a zigzag sequence containing coefficients from low to high value by the zigzag block. Finally, the coefficients are coded using the Entropy Encoder, in which Run-Length Encoding (RLE) algorithm and Huffman encoding are applied. The output is the compressed image JPEG file. The designs are tested using the 640x480 BMP image input file. The output files are compared with a golden file for correction.

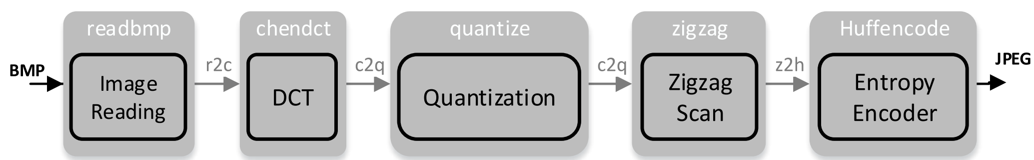


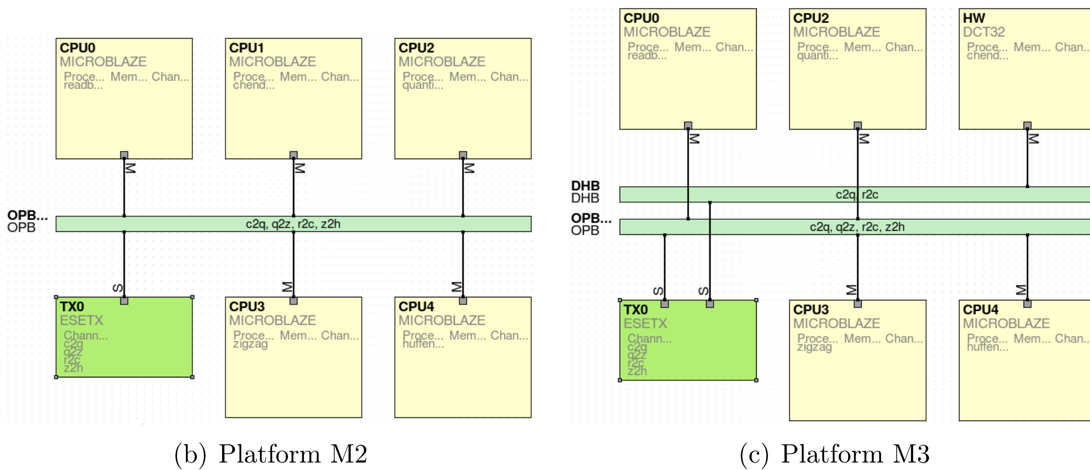
Figure 5.2 *JPEG encoder.*

Each block is implemented in independent C process communicating to each other via *send/recv* API functions mentioned above. The application is implemented in 3 platforms including:

- M1 (Figure 5.3(a)): pure software implementation on a single Microblaze processor. The processes transfers data via the FIFO memories with blocking read and write methods. The Microblaze includes the Xilkernel ROST implementing round-robin scheduling policy.
- M2 (Figure 5.3(b)): a platform includes 5 Microblaze processors, each implements one separate process. These processors connect to the Open Peripheral Bus (OBP) with interrupt synchronization. The processes communicate via uni-directional process-process message passing channels. The transducer resolves the traffic on the bus with round-robin scheduling policy.
- M3 (Figure 5.3(c)): the platform includes 4 processors MicroBlaze, and one

DCT hardware acceleration. The HW component is connected to the Double Handshake Bus (DHB), while the processors are connected to the Open Peripheral Bus (OBP). Both buses support interrupt synchronization. The transducer is required to transfer data between these two buses. It acts as a only slave on the two bus, and implements the round-robin scheduling policy. The communication channels between processes are also uni-directional process-process message passing channels.

(a) Platform M1



(b) Platform M2 (c) Platform M3

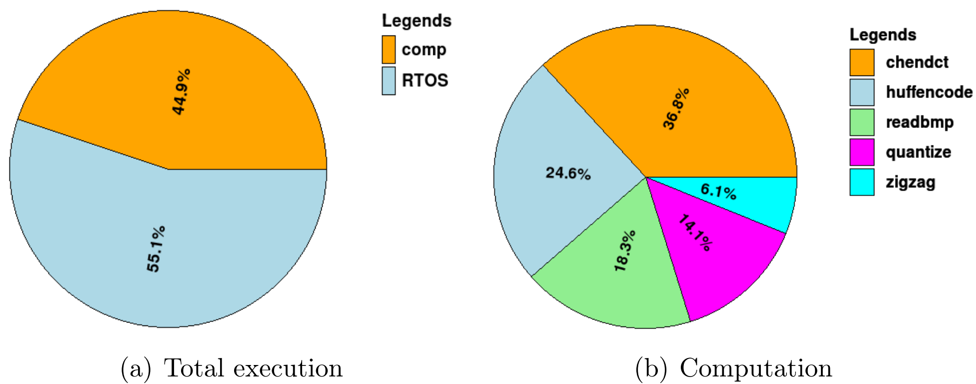
***Figure* 5.3** *JPEG encoder implementations in ESE*

The platform creations are simple and straightforward. The components are selected in the database, then dragged and dropped in the Main Window. The port and connection are easily made by few clicks. Process mapping is also facilitated by the GUI. Noticeably, ESE require all the code files must reside in one folder. Sub-folder containers or externally included files are unaccepted. Therefore, code flatten procedure must be done before mapping processes into the components. Each process is then assigned ports and channels, which are mapped to corresponding *send/recv* API functions. *tlmgen* and *tlmest* are then invoked to generate functional TLM and timed-TLM. These models can be simulated with user-configured options. The statistic graph can be created for each component after timed-TLM simulation to show execution and communication time of processes and functions associated with that component, as illustrated in Figure 5.4 and Figure 5.5. However, because each graph can show only one level, it is quite inconvenient to have a total view of highly hierarchical program with many levels of sub-functions.

The generation time of the functional TLM is quite fast, just a couple seconds, while simulation time is less than 1 seconds. Timed TLM is more complicated, but also takes in order of seconds for generation and simulation. The execution time estimated in clock cycles in Table 5.1 shows the significant performance enhancement from the simple platform M1 to the complex M3 with HW acceleration. The computation graph of M1 in Figure 5.4 confirms that DCT is the heaviest computing process, while the graphs in Figure 5.5(a) and Figure 5.5(b) show the bus traffic in OBP and DHB bus of M3 with round-robin scheduling. The total time spend on DCT IP also shown in Figure 5.5(c).

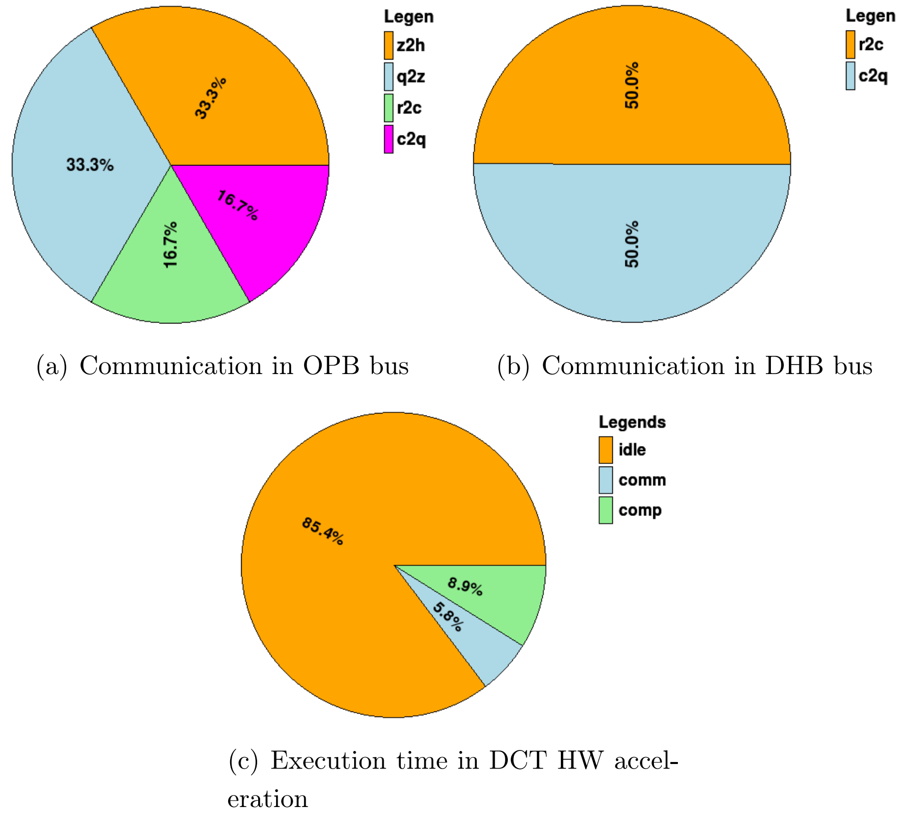
Table 5.1 *Design results.*

	Estimation Cycle
M1	2498088297 cycles
M2	531868634 cycles
M3	348062080 cycles



(a) Total execution (b) Computation

Figure 5.4 *M1 statistic performance graph.*



(a) Communication in OPB bus (b) Communication in DHB bus

(c) Execution time in DCT HW acceleration

Figure 5.5 *M3 statistic performance graph.*

In conclusion, the current version of ESE with easy usage and high automation does show competent results of one approach for front-end system design processes. However, due to incompleteness, obsolete and faulty, this is more merely suitable for demonstration than for any practical usage.

5.2 Daedalus

Daedalus is composed of several individual parts. The experimentation uses the newest versions, which are released around the year 2012 - 2013. Noticeably, the Sesame component currently has not been completed and publicly provided. Daedalus provides separate source code for each part, and a script file for automatic installation of the whole tool chain (with Sesame excluded). Although being released only couple years ago, several packages used by Daedalus are outdated, which imposed incompatibility problems in installation process. The framework is tested on Linux Mint 17.1, Centos 6.6, and Ubuntu 12.04, and only properly installed and executed on the last one.

PNgen part includes several tools (*c2pdg, pn, pn2adg, ...*) to convert forward and

backward amongst multiple model forms (SANLP, PDG, PPN, ...). ESPAM part includes *espam* tool, and SystemC library and simulator to generate timed-TLM files for simulation and project file for implementation (currently only Xilinx Platform Studio (XPS) project). Unfortunately, there is no detailed document about usage, requirement, options and constraints of these tools. Besides, The Daedalus doesn't provide a database of components in the installation. Only limited components are separately available in a sample application.

Daedalus doesn't provide GUI environment, so all development steps are done through commands and scripts. The most challenging part of Daedalus is that it requires programs to be specified in SANLPs form, which is not feasible for all applications and imposes difficulty on old code usages. This is also a constraint of applying Daedalus. Moreover, due to non-GUI environment, a platform structure and mapping descriptions must be manually written. Daedalus currently has not provided clear and complete guide for these. Given an application, platform structure and mapping descriptions, development process is highly automatic with invocation of a chain of tools to generate SystemC Timed-TLM or project files.

The experimentation of Daedalus is done with the simple application Sobel filter. The application includes five separate processes, as shown in Figure 5.6. The input image is first read into the program, then is applied gradient filter twice in different directions. The absolute value is computed and the result image is generated.

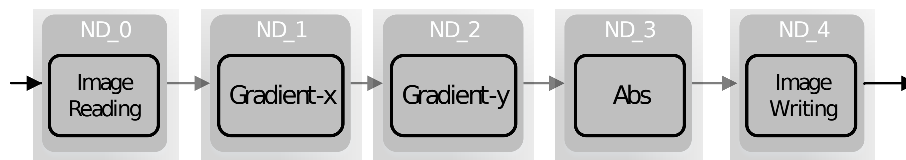


Figure 5.6 *Sobel filter.*

The program is mapped into two different platforms:

- M1: includes only one MicroBlaze processor implementing all processes
- M2: includes three MicroBlaze processors within which processes are distributed for maximum throughput: ND_0, ND_3, ND_4 are mapped to first processor, while ND_1 and ND_2, each is mapped to a different processor. The processors are connected using AXI crossbar switch.

The platform and mapping descriptions are shown in Figure 5.7.

```
<mapping name="myMapping">

   <processor name="mb_1">
       <process name="ND_0" />
       <process name="ND_1" />
       <process name="ND_2" />
       <process name="ND_3" />
       <process name="ND_4" />
   </processor>

</mapping>
```

```
<platform name="myPlatform">

<  processor name="mb_1" type="MB" data_memory="65536" program_memory="65536">
<     port name="IO_1" />
<  /processor>

<  network name="CS" type="AXICrossbarSwitch">
<     port name="IO_1" />
<  /network>

<  host_interface name="HOST_IF" type="ML605" interface="UART">
<  /host_interface>

<  link name="BUS1">
<     resource name="mb_1" port="IO_1" />
<     resource name="CS" p  ort="IO_1" />
<  /link>

</platform>
```

(a) Platform and mapping description for M1

```
<mapping name="myMapping">

   <processor name="mb_1">
       <process name="ND_0" />
       <process name="ND_3" />
       <process name="ND_4" />
   </processor>

   <processor name="mb_2">
       <process name="ND_1" />
   </processor>

   <processor name="mb_3">
       <process name="ND_2" />
   </processor>

</mapping>
```

```
<platform name="myPlatform">

<  processor name="mb_1" type="MB" data_memory="65536" program_memory="65536">
<     port name="IO_1" />
<  /processor>

<  processor name="mb_2" type="MB" data_memory="65536" program_memory="65536">
<     port name="IO_1" />
<  /processor>

<  processor name="mb_3" type="MB" data_memory="65536" program_memory="65536">
<     port name="IO_1" />
<  /processor>

<  network name="CS" type="AXICrossbarSwitch">
<     port name="IO_1" />
<     port name="IO_2" />
<     port name="IO_3" />
<  /network>

<  host_interface name="HOST_IF" type="ML605" interface="UART">
<  /host_interface>

<  link name="BUS1">
<     resource name="mb_1" port="IO_1" />
<     resource name="CS" p  ort="IO_1" />
<  /link>

<  link name="BUS2">
<     resource name="mb_2" port="IO_1" />
<     resource name="CS" p  ort="IO_2" />
<  /link>

<   link name="BUS3">
<     resource name="mb_3" port="IO_1" />
<     resource name="CS" p  ort="IO_3" />
<  /link>

 </platform>
```

(b) Platform and mapping description for M2

Figure 5.7 *Platform and mapping description.*

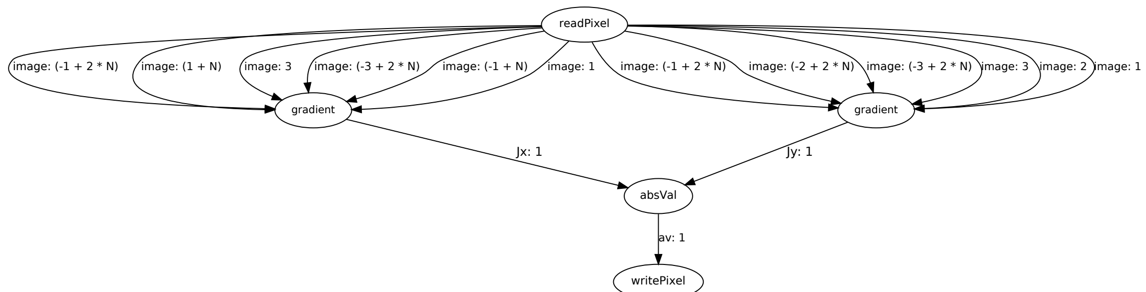


Figure 5.8 *Sobel PPN visual representation.*

Three tools from PNgen, namely *c2pdg, pn and pn2adg*, are sequentially invoked to convert the program in C SANLPs to XML-based PPN, which is accepted by *espam* tool. The resulted PPN can be visually represented, as illustrated in Figure 5.8. The

espam tool then converts the XML-based PPN to the timed-TLM SystemC, which can be simulated using Daedalus's simulator tool. As demonstrated in Figure 5.9, the simulator tool also provides various statistic performance information including execution time, read/write time, blocking time, utilization, .. of each PE in a platform.

```
P_1 finished at 4282087 ns

Statistics:
+---------+-------------+---------------+  +---------------------------------------+
| Process |     #clocks | Status        |  |         Utilization/Efficiency        |
+---------+-------------+---------------+  +----------------------------+----------+
|     P_1 |             |               |  |                        P_1 | 1        |
|         |     4282086 | Execution     |  +----------------------------+----------+
|         |           0 | Read          |  |      PPN Utilization/Efficiency:      |
|         |           0 | Write         |  +----------------------------+----------+
|         |           0 | Block on Read |  |                        PPN | 1        |
|         |           0 | Block on Write|  |   Sources and Sinks excluded | -nan   |
|         |           0 | Idle          |  +----------------------------+----------+
+---------+-------------+---------------+

+---------------------------------------+
|    Computation/Communication Ratio:   |
+----------------------------+----------+
|                        P_1 | inf      |
|                        PPN | inf      |
+----------------------------+----------+
```

(a) Statistic information of M1

```
P_2 finished at 2569825 ns
P_3 finished at 2569826 ns
P_1 finished at 2569831 ns

Statistics:
+---------+-------------+---------------+  +---------------------------------------+
| Process |     #clocks | Status        |  |    Computation/Communication Ratio:   |
+---------+-------------+---------------+  +----------------------------+----------+
|     P_1 |             |               |  |                        P_1 | 0.358    |
|         |      612966 | Execution     |  |                        P_2 | 0.1429   |
|         |      244608 | Read          |  |                        P_3 | 0.1429   |
|         |     1467648 | Write         |  |                        PPN | 0.2504   |
|         |      244608 | Block on Read |  +----------------------------+----------+
|         |           0 | Block on Write|
|         |           0 | Idle          |
+---------+-------------+---------------+  +---------------------------------------+
|     P_2 |             |               |  |         Utilization/Efficiency        |
|         |      122304 | Execution     |  +----------------------------+----------+
|         |      733824 | Read          |  |                        P_1 | 0.9048   |
|         |      122304 | Write         |  |                        P_2 | 0.3807   |
|         |     1591392 | Block on Read |  |                        P_3 | 0.3807   |
|         |           0 | Block on Write|  +----------------------------+----------+
|         |           6 | Idle          |  |      PPN Utilization/Efficiency:      |
+---------+-------------+---------------+  +----------------------------+----------+
|     P_3 |             |               |  |                        PPN | 0.5554   |
|         |      122304 | Execution     |  |   Sources and Sinks excluded | 0.5554 |
|         |      733824 | Read          |  +----------------------------+----------+
|         |      122304 | Write         |
|         |     1591393 | Block on Read |
|         |           0 | Block on Write|
|         |           5 | Idle          |
+---------+-------------+---------------+
```

(b) Statistic information of M2

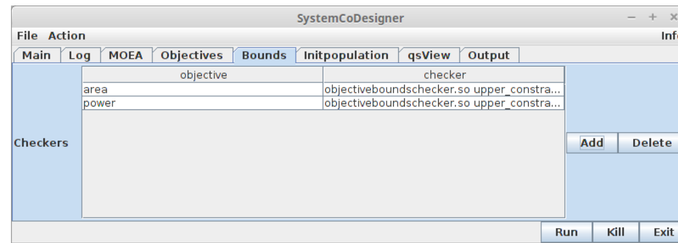
Figure 5.9 *Statistic information.*

In conclusion, Daedalus is still in testing phase, so most of the releases are immature and poor-documented. The wide application of the software is also limited by the input restriction and non-GUI environment. However, being amongst the most active projects with recent successive updates, the full and practical version with theses issues solved can be hoped to be released in the near future.

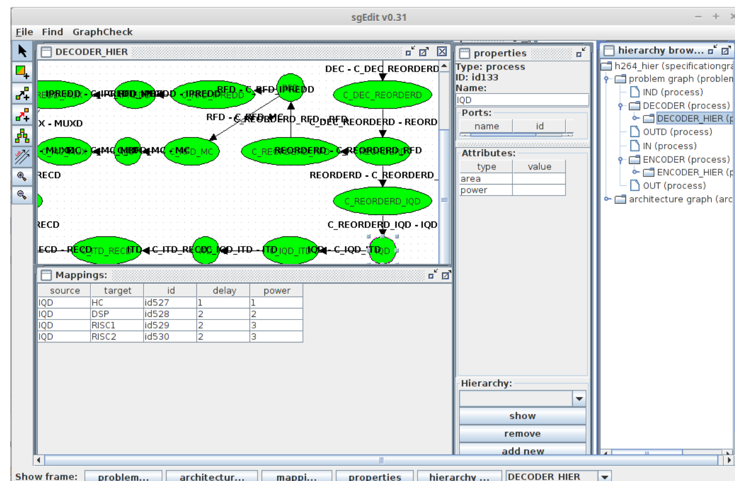
5.3 SystemCoDesigner

SystemCoDesigner released its only version 0.1 in 2006. The framework's developers provide a pre-built Java program, which can execute properly on various Linux OSes. The published version isn't hardly considered a complete design flow framework because the only function it offers is automatic design space exploration, which is implemented through three main tools:

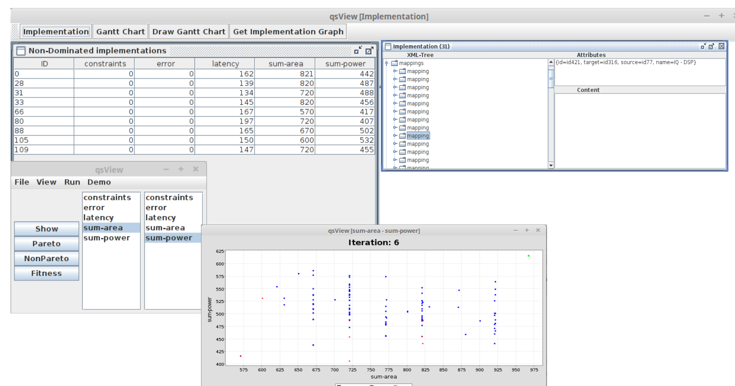
- *sgEdit* (Figure 5.10(a)): provides GUI environment for designers to visually describe an application and platform via nodes and edges, and specify mapping configurations between them. The mapping configurations include a set of feasible components each process can implement on along with associated delay, power and area attribute. The output of this process is served as input for automatic DSE tool.
- *systemcodesigner* (Figure 5.10(b)): functions as an automatic DSE tool. Designers can configure several parameters including settings of the MOE algorithm for automatic exploration (population size, mutation rate, polling delay, seed, allocation rate,...), a scheduling algorithm for latency evaluation (only listscheduler supported), a evaluation method for area and power (max, min, accumulation, product), upper and lower bound constrains for area and power. The setting of MOEA contains several theory-related fields, which are poorly documented, and may confuse designers with no knowledge of the algorithm.
- *qsView* (Figure 5.10(c)): shows all generated design points or only Pareto-points via several 2D graphs, and details of their mappings.



(a) Specification Editor *sgEdit*



(b) Automatic DSE tool *systemcodesigner*

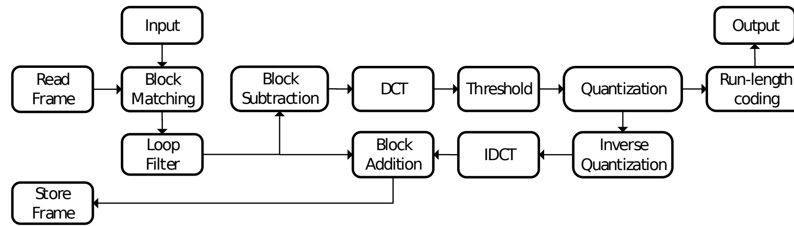


(c) Result viewer *qsViewer*

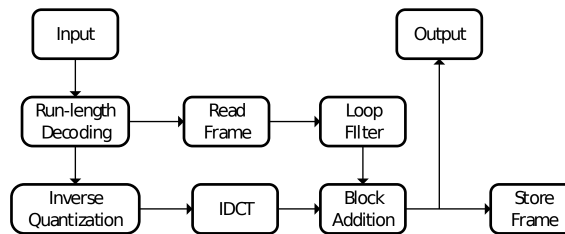
Figure 5.10 *SystemCoDesigner main tools.*

The framework is experimented with H264 video codec whose block diagrams are illustrated in Figure 5.11. The available resource is composed of two programmable processors RISC1, RISC2, one DSP processor, several custom HW components including frame memory (FM), dual ports frame memory (DPFM), Huffman coder (HC), subtractor-adder (SA), DCT/IDCT (DCT), and three bus types fast bus (FB), medium bus (MB), and fast bus (FB). The feasible mappings along with associated delays between each fundamental block of the application and each com-

ponent are shown in Table 5.2, power and area attributes of each component are shown in Table 5.3, and attributes of bus types are shown in Table 5.4.



(a) Coder



(b) Decoder

Figure 5.11 *H264 video codec.*

Designers only need to specify values for various parameters in *systemcodesigner* tool, and it will automatically generate optimal mappings based on the input specification file from *sgEdit* tools. A setting of MOE can be initially chosen with suggested values from the developers of SystemCoDesigner. Running time of the framework depends on the configured values, which can vary from a couple of minutes to several days. However, the result can be updated after every specified period, or each iteration, so designers can stop the process if there is an adequately good result. Total design points after 6 iterations are illustrated via latency vs. sum-area and sum-power vs. sum-area graphs in Figure 5.12. The red dot are non-dominated (Pareto) design point up to the current iteration. Reports of latency, area and power attributes of these Pareto designs are shown in Figure 5.13. The mapping details of 3 designs, which have minimum value of each attributes are listed in Table 5.5.

Table 5.2 *Feasible mappings of fundamental H264 video coder blocks.*

Block	Comp/Delay	Comp/Delay	Comp/Delay
Block matching	DSP/60	RISC/88	
Read frame	FM/0	DPFM/0	
Store frame	FM/0	DPFM/0	
Loop Filter	HC/2	DSP/3	RISC/9
Block subtraction	SA/1	DSP/2	RISC/2
DCT/IDCT	DCT/2	DSP/4	RISC/8
Threshold calculation	HC/2	DSP/8	RISC/8
Quantization/Inverse quantization	HC/1	DSP/2	RISC/2
Block addition	SA/2	DSP/2	RISC/2
Run-length coding/decoding	HC/2	DSP/8	RISC/8

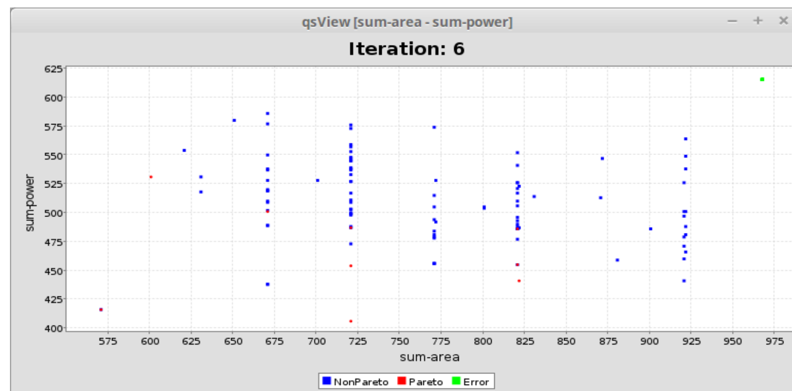
Table 5.3 *Attribute of PE components.*

Component	Area	Power
RISC	150	50
DSP	200	20
FM	20	10
DPFM	40	5
HC	50	3
SA	50	3
DCT	100	5

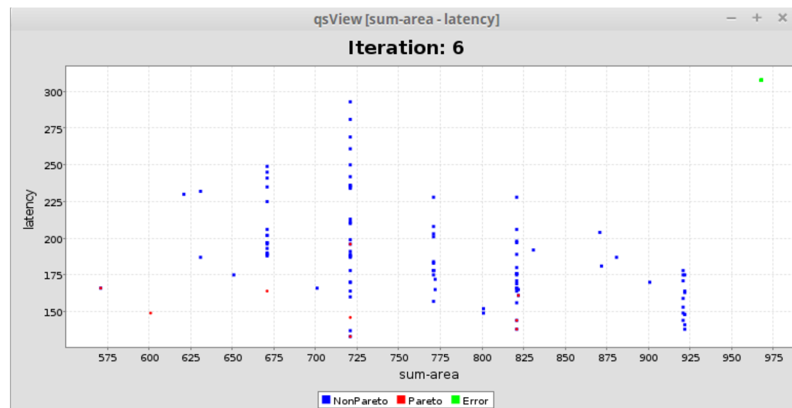
Table 5.4 *Attribute of buses.*

Component	Area	Power	Delay
SB	10	5	3 (1)
MB	20	10	2 (1)
FB	30	15	1 (1)

(1) delays of data transfer from Read frame block to block matching are 9, 6 and 3 for SB, MB and FB, respectively



(a) sum-power vs. sum-area



(b) latency vs. sum-area

Figure 5.12 *Total design point after 6 iterations.*

Non-Dominated implementations

ID	constraints	error	latency	sum-area	sum-power
0	0	0	162	821	442
28	0	0	139	820	487
31	0	0	134	720	488
33	0	0	145	820	456
66	0	0	167	570	417
80	0	0	197	720	407
88	0	0	165	670	502
105	0	0	150	600	532
109	0	0	147	720	455

Figure 5.13 *Latency, power, and area reports of Pareto design points.*

Table 5.5 *Pareto design points mapping.*

Block	Minimum Latency	Minimum Area	Minimum Power
Encoder			
Block matching	DSP	DSP	DSP
Read frame	FM	FM	DPFM
Block subtraction	SA	SA	SA
Threshold calculation	RISC2	HC	RISC2
Quantization	RISC1	RISC2	RISC2
Inverse quantization	DSP	RISC2	DSP
Block addition	RISC2	SA	SA
Store frame	DPFM	FM	DPFM
Loop Filter	DSP	RISC2	DSP
DCT	RISC2	DSP	DCT
IDCT	DSP	DSP	DSP
Run-length coding	HC	DSP	RISC1
Decoder			
Block addition	DSP	DSP	RISC1
Read frame	FM	DPFM	DPFM
Store frame	DPFM	FM	DPFM
Loop Filter	RISC2	DSP	RISC1
IDCT	DSP	RISC2	DSP
Inverse quantization	RISC2	RISC2	DSP
Run-length decoding	DSP	RISC2	RISC2

In conclusion, this release version of SystemCoDesigner framework does well to demonstrate an application of MOEA in automatic DSE, which is also the only function it can offer. The release is poorly documented with only short manual on the main website, and contains confusing MOE settings. Therefore, the framework is like a testing demonstration, and more feasible for the developers.

6. CONCLUSIONS

System design methodology has continually changed to cope with the rapid growth of the semiconductor industry as well as the increasing demand and pressure of the market. Starting with a traditional, sequential board-based method with long developing time, experience-based decision making, poor HW and SW development cooperation, and time-consuming manual work of most design steps, the design flow has been evolved to the modern, superior model-based design flow, which gives a complete solution for those old shortcomings. With the new executable system model - TLM, the model-based methodology can resolve most of the design work at the high-level abstraction. The system can be early evaluated through fast simulation for optimization and modification. Any change in the model is now much more effortless and immediately visible for further actions. This also enables automation in numerous design tasks (TLM synthesis, system model refinement, HW and SW generation,...), which greatly increases productivity and design-reuse, as well as reduces errors.

The Thesis presents academic frameworks and tools for the realization of the model-based design flows. For the very first design step - application modeling and simulation task, KTH and UC Berkeley propose ForSyDe and Ptolemy II, respectively. Both of them provide a heterogeneous modeling environment supporting combination of multiple MoCs, which is a must for capturing complexity of modern systems. UC Irvine, UC Los Angeles and UBS provide solutions for high-level HW synthesis from C application to HDL description in the system implementation step, namely NISC, xPilot, GAUT, respectively. Each tool carries special features to increase design quality, like CW datapath architecture and a compression technique in NISC, DRFM-based and pattern-based optimization mechanism in xPilot, and multi-mode architecture in GAUT. Other academic researches aim for creating complete, seamless design environments with high automation in several main design processes such as TLM synthesis and evaluation, model refinement, and HW and SW generation.These are SCE and ESE of UC Irvine, Metropolis of UC Berkeley, Daedalus of UVA, and SCD of FAU.

The study cases from these design frameworks and tools show promising results.

The developing time is greatly reduced with system models created, evaluated, and refined in a couple of hours to a few days. The implementations are also automatically synthesized and generated from libraries in order of seconds. Generated designs can be optimized with support of several multi objectives exploration algorithms. The design quality is also guaranteed by the accuracy of high-level system models estimation, which are shown less than at most 16% in comparison to the prototyping board.

However, most of the frameworks and tools are incomplete and immature. Numerous functions are still in a testing stage and only feasible for the developers. The biggest challenge is the constraint of input format of each tool, which limits the usage of these frameworks and tools as well as causes difficulty for designers. Besides, the frameworks and tools are developed independently without any universal standard, so it is almost impossible to combine them during a design process. The model-based methodology is quite new, and these are the very first attempts to realize it. There is still a lot of potential, and many improvements must be made to bring the methodology into practice.

BIBLIOGRAPHY

[1] S. Abdi, "Tlm automation for multi-core design," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*. IEEE Press, 2010, pp. 717–724.

[2] S. Abdi, Y. Hwang, L. Yu, H. Cho, I. Viskic, and D. D. Gajski, "Embedded system environment: A framework for tlm-based design and prototyping," in *Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on*. IEEE, 2010, pp. 1–7.

[3] S. Abdi, G. Schirner, I. Viskic, H. Cho, Y. Hwang, L. Yu, and D. Gajski, "Hardware-dependent software synthesis for many-core embedded systems," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*. IEEE, 2009, pp. 304–310.

[4] S. I. Association *et al.*, *International Technology Roadmap for Semiconductor (2011)*. available at http://public.itrs.net/.

[5] S. H. Attarzadeh Niaki, M. K. Jakobsen, T. Sulonen, and I. Sander, "Formal heterogeneous system modeling with systemc," in *Specification and Design Languages (FDL), 2012 Forum on*. IEEE, 2012, pp. 160–167.

[6] S. H. Attarzadeh Niaki and I. Sander, "Co-simulation of embedded systems in a heterogeneous moc-based modeling framework," in *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*. IEEE, 2011, pp. 238–247.

[7] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52, 2003.

[8] M. A. Bamakhrama, J. T. Zhai, H. Nikolov, and T. Stefanov, "A methodology for automated design of hard-real-time embedded streaming systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2012, pp. 941–946.

[9] M. Burton and A. Morawiec, *Platform based design at the electronic system level*. Springer, 2006.

[10] C. Chavet, C. Andriamisaina, P. Coussy, E. Casseau, E. Juin, P. Urard, and E. Martin, "A design flow dedicated to multi-mode architectures for dsp applications," in *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on.* IEEE, 2007, pp. 604–611.

[11] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xpilot: A platform-based behavioral synthesis system," *SRC TechCon*, vol. 5, 2005.

[12] W. Chen, X. Han, and R. Doemer, "Multicore simulation of transaction-level models using the soc environment," *IEEE Design and Test of Computers*, vol. 28, no. 3, pp. 20–31, 2011.

[13] W. Chen, X. Han, and R. Domer, "Esl design and multi-core validation using the system-on-chip environment," in *High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International.* IEEE, 2010, pp. 142–147.

[14] L.-y. Chiou, S. Bhunia, and K. Roy, "Synthesis of application-specific highly efficient multi-mode cores for embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 4, no. 1, pp. 168–188, 2005.

[15] J. E. Coffland and A. D. Pimentel, "A software framework for efficient system-level performance evaluation of embedded systems," in *Proceedings of the 2003 ACM symposium on Applied computing.* ACM, 2003, pp. 666–671.

[16] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "Platform-based behavior-level and system-level synthesis," in *SOC Conference, 2006 IEEE International.* IEEE, 2006, pp. 199–202.

[17] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays.* ACM, 2004, pp. 183–189.

[18] J. Cong, G. Han, and W. Jiang, "Synthesis of an application-specific soft multiprocessor system," in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays.* ACM, 2007, pp. 99–107.

[19] J. Cong and W. Jiang, "Pattern-based behavior synthesis for fpga resource reduction," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays.* ACM, 2008, pp. 107–116.

[20] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *Proceedings of the 43rd annual Design Automation Conference.* ACM, 2006, pp. 433–438.

[21] S. A. D. Gajski, *ESE Back End 2.0*, University of California, Irvine, 2006.

[22] D. Densmore, A. Donlin, and A. Sangiovanni-Vincentelli, “Fpga architecture characterization for system level performance analysis,” in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*. European Design and Automation Association, 2006, pp. 734–739.

[23] J. Eker, J. W. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, *et al.*, “Taming heterogeneity-the ptolemy approach,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[24] J. Falk, C. Haubelt, and J. Teich, “Efficient representation and simulation of model-based designs in systemc,” in *Proc. of FDL*, vol. 6, 2006.

[25] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded System Design: Modeling, Synthesis and Verification*. Springer Science & Business Media, 2009.

[26] B. Gorjiara and D. Gajski, “Fpga-friendly code compression for horizontal microcoded custom ips,” in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*. ACM, 2007, pp. 108–115.

[27] ——, “Automatic architecture refinement techniques for customizing processing elements,” in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*. IEEE, 2008, pp. 379–384.

[28] B. Gorjiara, M. Reshadi, P. Chandraiah, and D. Gajski, “Generic netlist representation for system and pe level design exploration,” in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. ACM, 2006, pp. 282–287.

[29] B. Gorjiara, M. Reshadi, and D. Gajski, “Generic architecture description for retargetable compilation and synthesis of application-specific pipelined ips,” in *Computer Design, 2006. ICCD 2006. International Conference on*. IEEE, 2007, pp. 356–361.

[30] ——, “Merged dictionary code compression for fpga implementation of custom microcoded pes,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 1, no. 2, p. 11, 2008.

[31] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich, “A systemc-based design methodology for digital signal processing systems,” *EURASIP Journal on Embedded Systems*, vol. 2007, no. 1, pp. 15–15, 2007.

[32] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith, "Systemcodesigner: automatic design space exploration and rapid prototyping from behavioral models," in *Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 580–585.

[33] J. Henkel and S. Parameswaran, *Designing embedded processors: a low power perspective*. Springer Science & Business Media, 2007.

[34] Y. Hwang, S. Abdi, and D. Gajski, "Cycle-approximate retargetable performance estimation at the transaction level," in *Proceedings of the conference on Design, automation and test in Europe*. ACM, 2008, pp. 3–8.

[35] T. D. Hämäläinen, "Lecture 2: System design flow," in *System Design*. Tampere University of Technology, 2015.

[36] J. Keinert, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, M. Meredith, *et al.*, "Systemcodesigner-an automatic esl synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 1, p. 1, 2009.

[37] B. Kienhuis, E. F. Deprettere, P. Van Der Wolf, and K. Vissers, "A methodology to design programmable embedded systems," in *Embedded processor design challenges*. Springer, 2002, pp. 18–37.

[38] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: Deriving process networks from matlab for embedded signal processing architectures," in *Proceedings of the eighth international workshop on Hardware/software codesign*. ACM, 2000, pp. 13–17.

[39] E. A. Lee and I. John, "Overview of the ptolemy project," 1999.

[40] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor system design, programming, and implementation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 3, pp. 542–555, 2008.

[41] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere, "Daedalus: toward composable multimedia mp-soc design," in *Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 574–579.

[42] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *Computers, IEEE Transactions on*, vol. 55, no. 2, pp. 99–112, 2006.

[43] C. Ptolemaeus, *System Design, Modeling, and Simulation: Using Ptolemy II.* Ptolemy. org Berkeley, CA, USA, 2014.

[44] D. Rainer, G. Andreas, P. Junyu, S. Dongwan, C. Lukai, Y. Haobo, A. Samar, G. Daniel D, *et al.*, "System-on-chip environment: A specc-based framework for heterogeneous mpsoc design," *EURASIP Journal on Embedded Systems*, vol. 2008, 2008.

[45] M. Reshadi and D. Gajski, "A cycle-accurate compilation algorithm for custom pipelined datapaths," in *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis.* ACM, 2005, pp. 21–26.

[46] E. Rijpkema, E. F. Deprettere, and B. Kienhuis, "Deriving process networks from nested loop algorithms," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 165–176, 2000.

[47] I. Sander, "System modeling and design refinement in forsyde," Ph.D. dissertation, Royal Institute of Technology, 2003.

[48] I. Sander, A. Jantsch, and Z. Lu, "Development and application of design transformations in forsyde," *IEE Proceedings-Computers and Digital Techniques*, vol. 150, no. 5, pp. 313–320, 2003.

[49] T. Schlichter, M. Lukasiewycz, C. Haubelt, and J. Teich, "Improving system level design space exploration by incorporating sat-solvers into multi-objective evolutionary algorithms," in *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on.* IEEE, 2006, pp. 6–pp.

[50] T. Stefanov, E. Deprettere, and H. Nikolov, "Multi-processor system design with espam," in *Hardware/Software Codesign and System Synthesis, 2006. CODES+ ISSS'06. Proceedings of the 4th International Conference.* IEEE, 2006, pp. 211–216.

[51] M. Streubühr, J. Falk, C. Haubelt, J. Teich, R. Dorsch, and T. Schlipf, "Task-accurate performance modeling in systemc for real-time multi-processor architectures," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings.* European Design and Automation Association, 2006, pp. 480–481.

[52] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, "A framework for rapid system-level exploration, synthesis, and programming of multimedia mp-socs," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2007, pp. 9–14.

[53] J. Trajkovic and D. Gajski, "Automatic data path generation from c code for custom processors," in *Embedded System Design: Topics, Techniques and Trends*. Springer, 2007, pp. 107–120.

[54] A. Turjan, B. Kienhuis, and E. Deprettere, "Translating affine nested-loop programs to process networks," in *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 2004, pp. 220–229.

[55] *NISC toolset user guide*, University of California, Irvine, July 2008.

[56] *Embedded System Environment: ESE Version 2.0.0 - User Manual*, University of California, Irvine, Sep 2008.

[57] S. Verdoolaege, H. Nikolov, and T. Stefanov, "Pn: a tool for improved derivation of process networks," *EURASIP journal on Embedded Systems*, vol. 2007, no. 1, pp. 19–19, 2007.

[58] E. Zitzler, M. Laumanns, L. Thiele, E. Zitzler, E. Zitzler, L. Thiele, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," 2001.