Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos. Raportti 11
Tampere University of Technology. Department of Software Systems. Report 11

Mika Siikarla

Applying the DReAMT Model Transformation Approach in Two Studies

TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos. Raportti 11
Tampere University of Technology. Department of Software Systems. Report 11

Mika Siikarla

Applying the DReAMT Model Transformation Approach in Two Studies

Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos
Tampere 2010

ISBN 978-952-15-2381-6
ISSN 1797-836X

Abstract

Decision Reusing Approach for Model Transformations (DReAMT) is a lightweight approach for developing interactive model transformations. The approach consists of an iterative model transformation development process and the use of decision modeling. There is also a proof-of-concept tool set to support the approach.

In this technical report we describe two studies where the DReAMT approach and tool were used. In the first study we developed a model transformation to be used as a component in a process modeling tool. In the second study we began to develop a model transformation to support the creation of model checking rules for a flexible modeling tool. The first study was carried out with Solita Oy and the second with Trinity research team from Tampere University of Technology.

Evaluation of the studies shows that (i) the DReAMT approach can be used to develop model transformations; (ii) interactive model transformations are flexible; (iii) decision modeling makes automatic decision reuse possible; and (iv) the DReAMT tool can be used by people other than its author.

1 Introduction

Decision Reusing Approach for Model Transformations (DReAMT) [1] is a lightweight approach for developing interactive model transformations. There is also a proof-of-concept tool to support using the approach. The main principle of the approach is that a model transformation is developed in iterations, starting with an incomplete transformation that contains a lot of human interaction. The model transformation is gradually refined and the human interaction is made more structured in each iteration and finally possibly even automated fully.

In the beginning of the development of a model transformation–or any software– there is typically very little known with certainty about the requirements. The understanding and certainty grows during the development when new situations are encountered and good and bad solutions are tried. Iterative development makes it possible to start with a model transformation that helps the software designer with just a few of the best understood tasks and grow the transformation as the understanding of the requirements grows.

In the first iterations the model transformation may be little more than a pile of model editing scripts, some of which are launched automatically and some manually. The transformation may have to also allow unrestricted editing of the model so that the designer can complete all of their design tasks. With improved understand in the next stage of iterations the transformation may resemble a wizard. Most of the unrestricted model editing is replaced by a few chosen, tool guided questions using the domain terms. In the later stages the model transformation may be automatic or contain just a couple of human decision points.

In this technical report we describe two research studies where we applied DReAMT and evaluated the approach and the tool. In the first study the DReAMT tool was used as a component in a business process modeling tool. A DReAMT model transformation was used to transform a business process that was expressed as an activity diagram into an XML-form understood by a process and task engine. In the second study the DReAMT process was used to develop a model transformation that assists in creating model (in)consistency rules based on two metamodels. The first study was carried out with Solita Oy and the second with Trinity research team from Tampere University of Technology.

2 DReAMT

The DReAMT approach includes an iterative model transformation development process, a model transformation language and a model for expressing human decisions. The process defines the roles of Design Phase Expert, Transformation Architect and Transformation Programmer as well as the responsibilities and skill requirements for the roles. The process is centered around artefacts, which are used in communication between the roles. The requirements specification is captured in *transformational patterns*, the architecture and design in a *model*

transformation definition and the final result in the *model transformation implementation*.

The cyclical model transformation development process is illustrated in Fig. 1. The artefacts created and refined in the process are shown as wide boxes in the middle of the picture. A shaded area connects an artefact to the role that is primarily responsible for it. Application development consists of several design phases and there is a separate instance of the DReAMT process for each design phase.

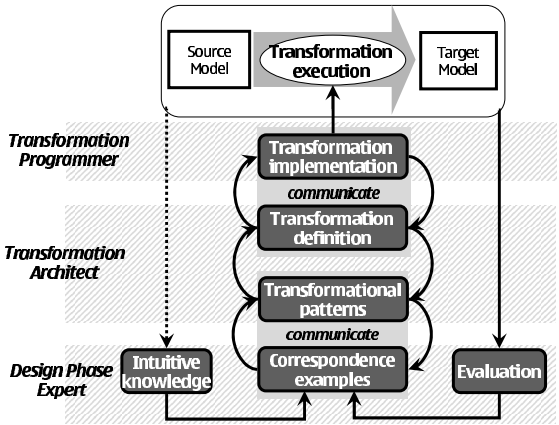

Fig. 1: Transformation development process and its roles [1, Fig. 1]

The model transformation language is based on patterns and graph grammars. Each pattern implementation in the model transformation implementation roughly corresponds to one transformational pattern in the requirements specification. So, the specification and the implementation have the same basic unit of modularity, which helps contain the effects of incremental changes to the specifications.

The human decision modeling gives the Transformation Architect and the Transformation Programmer a way to define the context of a decision. A decision context is the set of model elements and previously made decisions that affects this decision. The decision context can be used to automatically reason about the decision's validity when the source model has been updated.

The DReAMT tool, which supports the DReAMT approach, is actually two tools, the interactive model transformation execution environment (DIMTEE) and the model transformation compiler (DMTC). The execution environment is built on top of a task-based role modeling tool MADE [2]. The compiler is a stand-alone program.

MADE is a versatile tool that was originally developed for framework specialization. DReAMT uses MADE to handle the user interface and to manage and perform model editing tasks, which is only a small part of the functionality

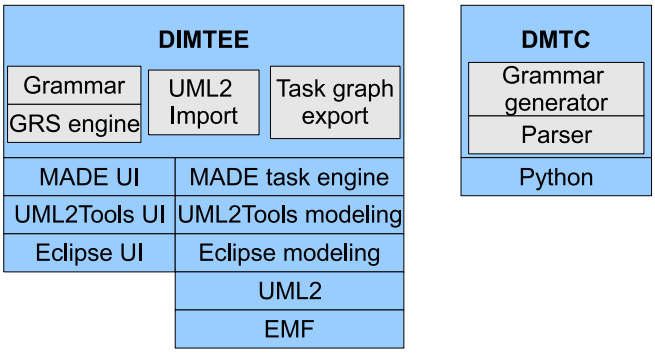

Fig. 2: DIMTEE and DMTC technology stacks

available. MADE itself is integrated with a UML modeling tool. The current version is integrated with UML2Tools [3], which is built on top of Eclipse [4] and uses the EMF-based (Eclipse Modeling Framework) UML2 component.

DIMTEE is integrated with MADE and the Eclipse UML2 component. The UML2 component is based on the Eclipse Modeling Framework (EMF). MADE is integrated with UML2Tools, which provides diagram editors on top of UML2. DMTC uses only common Python libraries. The components DIMTEE and DMTC consist of (light boxes at the top) and depend on (dark boxes at the bottom) are shown in Fig. 2.

When the user launches a model transformation he gets a list of tasks. Performing one task may bring new tasks to the list. When all the tasks have been performed, the transformation is finished.

It is not within the scope of this report to explain the DReAMT approach or tool in more detail. However, the reader does need some knowledge of how the approach and the tool are used in order to understand the application sections. We suggest that a reader who wants to know more about the model transformation development process [1], the model transformation language [5] [6], modeling human decisions [6] or the tool [7] reads the author's other publications.

3 Solita Process Language Modeling Tool

3.1 Solita Process Language

This study was conducted in cooperation with Solita Oy. They were developing a business process management system for an external client company. The goal of the study was to explore how process modeling could be used at Solita Oy by developing a process modeling tool prototype. The DReAMT tool was used to transform a process model into a proprietary XML-based process language used by the business process management system.

The business process management system runs small business processes, such as handling a product order. Despite the small size, a process may run for several

days or even months. The system contains a process engine, which starts and stops instances of processes manages them when they are running. The system receives events from business systems and directs the running process instances, if necessary.

A process description contains declarations of *tasks*, which are performed by humans. A TaskBuilder service within the process engine creates and initializes tasks and assigns active tasks to users. An external task management system handles the user responses and updates the state of the tasks accordingly. The process engine monitors the active tasks and reacts to changes in their states. Task types are extensions to the TaskBuilder service.

A process description is essentially a state machine. It contains states, which are called *nodes*, and *transitions* between the states. A transition can have a condition (a *predicate*) that depends on, for example, external events and timeouts. Exactly one of the states is active at any given time during the execution of a process instance. When a predicate on any of the outgoing transitions is fulfilled, the target of the transition becomes the active state. Even if more than one condition becomes true at the same time, only one of the transitions is followed.

There are several types of nodes. A so called *workphase node* has tasks associated with it. When the state becomes active, the tasks are activated and the task management system allows users to perform them. A *fork node* contains descriptions of subprocesses, which are executed in parallel. When all the subprocesses have finished, the outgoing transitions from the fork node may be triggered. The execution in the main process does not fork. There is only one active state in the main process (the fork node) and one in each of the subprocesses.

The process instance also has data that can be used during the execution, for example the identifier of the product order being processed. The transition predicates can refer to the process instance's data.

The process description is given in a proprietary XML format. The majority of the elements in the XML language are straight-forward, such as *node*, *transition* and *predicate*. However, the semantics of some XML elements and XML attributes, for example *scheduler* and *class*, are not immediately clear from their name.

A process description in the XML form is passed to the business process management system, which creates a Java code component. When a process instance is started, the Java objects required to hold the process instance's state are created. In the beginning the new process instance's active state is its start node.

3.2 Setup for the Study

The core of the study was carried out by a researcher (the author of DReAMT) and a research assistant from Tampere University of Technology (TUT) and a process designer and the author of the process engine from Solita Oy. Several other people from both organizations participated in smaller roles. Most of the work was carried out in about a year's time in 2009–2010.

The process designer had participated in the requirements capture for some of the business processes, so he knew them very well. He knew the basics of the process language and its XML format, but had not written any process descriptions in XML. The author of the process engine had not been involved in the requirements capture and had only seen a few examples of the real business process specifications. As the lead designer, he knew the process language and its XML format thoroughly. Both the process designer and the author of the process engine had used Eclipse before, but not for modeling. Neither of them was familiar with the UML activity diagram notation.

The process modeling tool was built by customizing UML2Tools and MADE and by writing a DReAMT model transformation. The research assistant acted as the main tool developer and did most of the customization and wrote the model transformation. He had no prior experience of model transformations or DReAMT. The researcher supervised the study and did some modifications to MADE and DReAMT. The process designer and the process engine's author provided information about the process language and assessed the process modeling tool.

The process modeling tool was meant to be used by the people who write the XML process descriptions based on business process specifications. The process modeling language was defined as a profile for a UML activity diagram. The profile defined stereotypes and attributes for the stereotypes. The stereotypes were used to distinguished between the various node and transition types. Node type specific details, for example the name of a Java class used for callbacks, were stored in the extended attributes of the stereotypes. The syntax and visual appearance of activity diagrams was used, but the semantics were different.

In the process modeling too, the user would first draw the process model using the UML editor and then launch a model transformation to generate the XML format. The process model is mostly just a visualization of the process, so the model transformation from an activity diagram to the XML format is a syntactic translation. The process modeling tool developer used the DReAMT model transformation language and tool to create the model transformation.

Because the primary goal was to learn more about the suitability of process modeling for Solita Oy and not just to create a tool, the plan was to create an initial tool version early and improve it in cycles. The first version would be used just for gathering experiences and feedback to find better requirements. Also, collecting the process patterns can be started when some version of the tool is available.

3.3 Building the Process Modeling Tool

The process modeling tool developer started by customizing the tool platform. He added support for profiles and stereotypes to MADE, fixed some bugs in the interaction with activity diagrams and so on. He also studied about graph-rewrite systems and model transformations.

The researcher added the ability to process activity diagrams to the UML2 Import component in the DReAMT tool. This was easy, because it only involved

modifying the translation model, which is used to generate the import code. The researcher also helped the process modeling tool developer learn about the DReAMT model transformation language and about using the DReAMT tool.

After the training for using the model transformation language, the tool developer wrote the model transformation independently. The DReAMT model transformation development process was not used, because the model transformation was merely a mapping. The activity diagram and the process language had nearly identical structure by design, so the model transformation mostly just recreated the structure using different node types.

The initial version of the process modeling tool was based on many guesses about the process language. Technical documentation about the business process management system and especially the process language had been sent to TUT and it had been studied there. The documentation was quite technical and had been cleaned of references to Solita's client. It was difficult to understand some aspects of the process language, because the documentation was on a low level and lacked a real context. The syntax was clear from the XML schema, but the semantics of, e.g. the different kinds of nodes were not clear.

In principle it would have been possible to design a visual language based on the syntax alone. However, the aim was for humans to read and write processes in the language and it was important to know the meaning of concepts and not just their form. In order to be suitable, a visual language needs to emphasize the important and hide the less important features.

At this point the process designer and process engine's author were not yet directly involved. They were working on other projects, but questions were frequently directed to them, because they knew the most about the business process management system. Communication was done via email, which is always challenging. We do not mention these obstacles to chronicle the rare and exotic misfortunes that befell us. We mention them, because such events are common in real projects in real environments.

The modeling tool was demonstrated to the process designer and the author of the process engine. They quickly discovered mistakes in the representation of the processes. The errors were caused by the initial misunderstandings and lack of knowledge about the semantics of the process language. When the participants were in the same location, communication was, of course, much more efficient than by email. Many of the misunderstandings were corrected in the same session. Despite its shortcomings the first tool version served its purpose of generating a lot of feedback.

For the second version of the process modeling tool, the tool developer made modifications to the process modeling language, i.e. the activity diagram profile, and to the XML generation. He also made some bug fixes and general improvements to the code. A process model now contained all the same information as a process description in the XML form. The workflow was visible as activities and flows, that were stereotyped to mark the various node and transition types. All the details, such as Java class names for callbacks, were stored in attributes of

the stereotypes. In UML2Tools these extended attributes of the elements were only visible when viewing an element's properties.

Note that this is not a model marking [8, p. 3-6] approach. The stereotypes and details are part of the process description and they are used by a process writer. They are not additional markings used only in the model transformation.

At the same time, the writing of processes had started at Solita Oy. Because these were the first real processes written in the process language, new requirements for were naturally discovered. As a consequence, changes were made to the process engine's capabilities and the process language to accommodate to the new needs.

The process modeling tool still produced process descriptions in the old XML format. It was presented and delivered to Solita Oy with a user manual. Unfortunately the tool was too difficult to be used without the presence of the tool developers, despite the manual and a training session. There were also some instability issues with the integration of UML2Tools and the rest of the process modeling tool. The poor usability and technical problems prevented independent use of the tool at Solita Oy.

The third version of the process modeling tool produced process descriptions in the new XML format. A lot of effort was spent to improve the usability and the stability, but the problems never were satisfactorily solved in the scope of the project. Even with additional training, the tool was not mature enough to be used independently. It was therefore decided that the researcher, being familiar with the process modeling tool and the workarounds to its problems, would use the tool to model two real business processes. The process designer chose two processes he considered to be among the most complicated ones.

The two business processes were described in process specification documents that contained a textual description of a business process and a picture in an *ad hoc* workflow notation. It showed the structure of the process, conditions on transitions and the tasks. The specifications had been written by different people, but the pictures were similar and closely resembled the activity diagram notation used in the process modeling tool.

The notation used in the pictures was not formalized. The pictures were meant only to augment the text. They contained small inconsistencies in the notation, for example the difference of branching and forking was not clearly marked and some transitions had been accidentally left out.

The researcher modeled the two processes. The activity diagram representations closely resembled the pictures in the process specification documents. The two process models, the produced XML and the tool itself were presented at Solita Oy and they were discussed.

A modified activity diagram for one of the processes is shown in Fig. 3. The names of the process, the nodes and the tasks have been changed for confidentiality reasons. The structure in the diagram is not changed. The rounded boxes are nodes and the connecting arrows are transitions. The texts on the transitions are predicates. The rectangles are tasks and the arrows with the text "using" connect a workphase node to the tasks it activates.

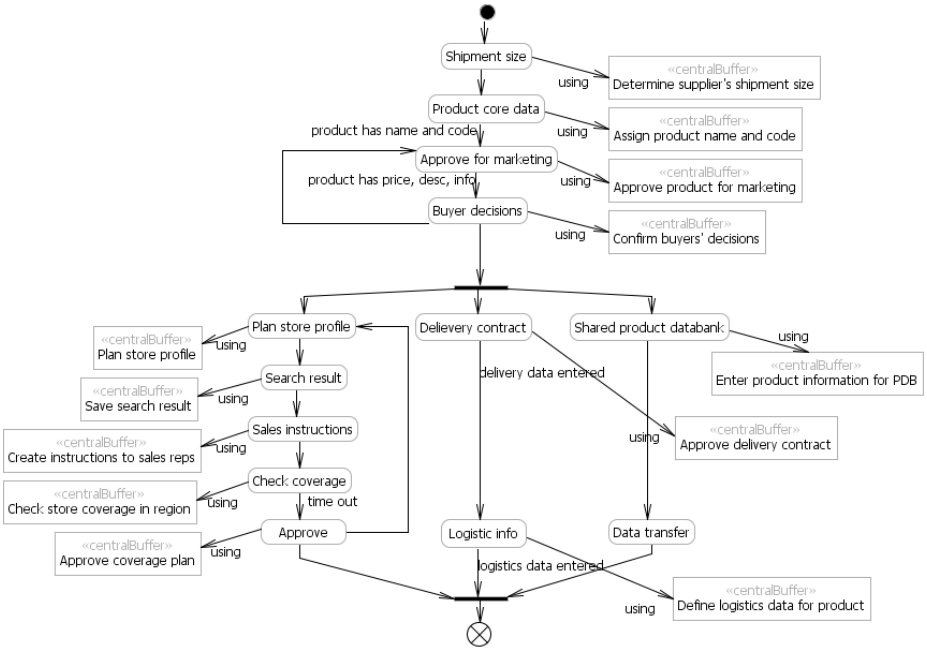

Fig. 3: Process model for creating a new product

The modified process handles the creation of a new product type for a wholesale company who sells and delivers products to other companies or franchises. The process starts at the top and the normal workflow proceeds down. The top part of the process handles the creation of the product type into the wholesaler's information system. The bottom part consists of three parallel paths. The leftmost path deals with instructing regional salesmen to which companies they should try to sell the new product. The middle path processes delivery contracts and logistics with the supplier. The rightmost path is for entering the product's information into a product databank shared by the wholesaler and its clients.

3.4 Feedback from Solita

The study proceeded in iterations and throughout the study the process designer and the author of the process engine gave feedback on process modeling and its suitability for Solita Oy in this context. During the study they gave feedback in discussions and in email correspondence and after the study they gave a free form summary. The process designer and the author of the process engine were given a short list of topics to cover in the summary and they were encouraged to include any other comments they wished.

Because the model transformation is only a small part in the process modeling tool, the feedback is not directly about the DReAMT tool. However, the process

designer and the author of the process engine make interesting observations, which we find relevant from the point of view of developing model transformations, modeling language design and application of modeling (and thus also applying MDE). For completeness, we provide most of the generic feedback as well, although in an abridged form.

According to the feedback, the visual notation for the process model makes the structure of the business process easy to understand. Examining details, however, is difficult. The visual notation did not contain or highlight non-essential aspect, but some relevant features, for example attributes of nodes, were hidden. Making all the details visible would clutter the process and obscure the process structure. Some parts of the UML activity diagram notation were considered difficult to understand.

Understanding of the uses of process modeling grew during the study and many initial assumptions were discovered to be wrong. The process modeling tool produced and delivered did not abstract from the XML, that is, it did not remove the need to manipulate and understand the details present in the XML. Either the details need to be added into the visual process model or they have to be added to the XML after it has been generated. Modifications to the XML would be lost if the XML was generated again. For these reasons, the process modeling tool is not very useful for a process writer, even if the technical problems were fixed.

The model transformation is not very visible to the process modeling tool user, so there is not much feedback about it. It correctly translated the information in the extended activity diagram form into the XML form. Optional attributes for nodes could be set and default values changed during the XML generation. However, changing the values requires detailed understanding of the XML format.

The process modeling tool served its purpose in helping explore process modeling in Solita's context, but it turned out that the need for a modeling tool are different from the initial guesses. Therefore the process designer and the author of the process engine did not consider a process modeling tool with these features to be suitable for use at Solita Oy.

With the experience and increased understanding of the potential and uses of process modeling in Solita's context, the author of the process engine and the process designer described what kind of a process modeling tool would be most useful. The most important purpose for a process modeling tool from Solita's perspective is to make creating business processes easier, so that the user does not need to know the details of the process language.

A process modeling tool should be well aware of the process language, so it could help and guide the user with difficult actions. The tool could guide the user, e.g. by requiring mandatory attributes to be given and suggesting default values for optional ones. This information is already in the XML schema, and could perhaps be used in the process modeling tool to adjust its behaviour.

Such a tool would complement direct XML editing instead of replacing it. The graphical notation and the UML2Tools editor work well for creating and

modifying the structure of the process, but the details need to be modifiable as well. The structure of the process could be viewed and modified in visual form and details added into the XML. A round-trip engineering functionality would allow propagating changes between the process model and the XML without losing information. It should also be possible to start with either format and generate the other.

A process modeling tool should support incremental process development, so that a sketch of a process could be gradually refined and augmented with the necessary details. The tool could maybe be used in the requirements capture phase to draw an initial sketch with the bare amount of details. The sketch would then be passed on to a process writer more as a specification than as an incomplete process description. It is not certain that such use of the tool would reduce the total effort, but it would help people with different skill sets to participate in implementing a business process.

The appearance of various visual elements should be customizable. This would help visually distinguish between nodes whose type is the same, but that have some specific values for its attributes. For example, a node that triggers an asynchronous external operation could look different from normal nodes.

3.5 Observations and Conclusions

Because the model transformation was such a clear translation from one format to another, and especially since the business process model was shaped following the XML-format, the DReAMT process was not used. So, this study evaluates only the use of the DReAMT model transformation language and the DReAMT tool for developing model transformations.

The process modeling tool developer designed and wrote the first version of the model transformation and each update on his own. The author of DReAMT participated in designing the process model and visualizations for several of the process language concepts and provided a little assistance in bug hunting, but did not write any pattern implementations or application rules.

This was the first time the process modeling tool developer used DReAMT. He had no problems with writing the implementation in the model transformation language nor with using the model transformation compiler DMTC. There is no real debugging support, which would be vital in a finished production quality tool set. So, there were a few bug hunt sessions, where he needed help.

The researcher added support for UML activity diagram elements into the DReAMT tool. This was very easy due to the translation model that is used for generating the model import component.

We conclude that the DReAMT model transformation language and tool can be used with little initial training and that extending it to work with new types of UML diagram notations is easy.

The goal of the study was not to apply MDE to business process implementation at Solita Oy. However, exploring the possibilities for modeling and code generation are close to what could be a first stage in applying MDE. We there-

fore think that some observations made in this study could be generalized for many MDE attempts.

In this case, there was no particular process of how a business process specification becomes a business process implementation. The natural language specification is just handed to a process writer who somehow writes an XML file. No one knew initially what would be the best place for modeling, what should be modeled or how. The understanding of these issues slowly grew during the development of the process modeling tool. One of the modeling languages—the XML format for the business processes—evolved during the development. Many initial guesses were wrong, including such fundamental issues as the best target group for the modeling and tooling.

Such challenges are typical to software development and we have argued that they are typical for its small subset, model transformation development, too. If models and high quality model transformations had been developed based on only these bad guesses, it would have been a disaster. A thorough analysis prior to developing the model transformations would have delayed the start of implementing the business processes and would not have helped in the end, because some problems were discovered only while the real business processes were being implemented. We see this as support to our claim that model transformation development should happen iteratively and in parallel to development of the application (in this case business process implementations).

We also take this opportunity to examine the reasons the process modeling tool was not found useful. The tool should have allowed crafting business processes easier than with XML. However, the business process model contained all the same information as the XML format. Nothing was abstracted away. Although the overall structure of the process was easier to see, the details still needed to be added.

The modeling language was just a different syntax for the same content. That does not raise the abstraction level. Just adding a proper "model" or a visualization does not help on its own. The abstraction level must be raised by leaving out details, in order to gain much benefits. The process description already had a structured well-defined form, so having a (UML) model did not add anything new.

Another reason for the unsuitability of the process modeling tool was that the values of some properties were much more important than expected. The process engine has many extension points and some properties of nodes determine the extension to use. Two nodes with different values in such a field could be semantically very different and hiding this distinction in the visual notation actually makes the process more difficult to understand. This is the reason why the process designer and the author of the process engine mentioned customizable appearance for subtypes of elements.

We think a better use for the process modeling tool would have been in the requirements capture, where a process model and its visualization could replace the sketches made with a regular drawing program. In that way, it might have even been possible to use the first process not only to visualize the process struc-

ture but also to simulate some usage scenarios. The business experts involved in crafting the business process specification would have been able to run previously defined or *ad hoc* usage scenarios to look for any obvious logical mistakes.

4 A Flexible Modeling Tool System

4.1 Trinity

This study was conducted at Tampere University of Technology. The Trinity project had developed flexible modeling tool system called Trinity, which allows manipulating incomplete and even inconsistent models. The Trinity project was going to build a component that informs a modeling tool user how the model is inconsistent with its metamodel. The goal of the study was to develop a process and tool support for creating inconsistency rule sets based on the differences between the metamodel and what actually is allowed in the tool. The DReAMT approach was going to be used to develop a model transformation to assist in creating such rule sets.

For the purposes of this study, the Trinity system consists of a model repository and integrations to various modeling and reporting tools. The model repository is a relational database and the tool integrations communicate with it using an object-relational mapping component. The integrated tools themselves can vary from modeling environments like Eclipse UML2 to drawing tools with some diagram support, e.g. Microsoft Visio, and reporting tools such as Microsoft Excel or Word.

In reality Trinity contains much more. The model repository is distributed, not local and centralized; tool integrations can communicate with each other using an agent architecture; models can be annotated, reviewed and linked to each other; and models can be versioned. For the purposes here, however, the simplified view of Trinity suffices.

The tool user does not directly edit the model with the tool, instead he edits a view of the model. A view is a tool specific representation of a full or a partial model and can contain graphical and textual elements and information such their size, position and colour. The elements in the view are also linked to the elements in the model. Both the view and the model are stored in the model repository. The model repository also contains a metamodel both for the model and for the view. There can be more than one view to a model and the views can be for different tools.

Trinity is integrated to a modeling or reporting tool by building a component specifically for that tool. The component observes the tool user's actions, e.g. drawing an element or connecting elements, and interprets their effects on the view and the model. The changes are reported to the object-relational mapping component, which stores them in the repository.

4.2 Incomplete and Inconsistent Models

Trinity allows incomplete and inconsistent models because of the view that modeling is more than just entering the finished model into a tool. Modeling is pri-

marily a creative design activity and the role of the model changes during it. Especially in the early stages the model may be used for communicating and visualising ideas and thoughts. Such sketching should not be restricted by demands of completeness and full compliance with the modeling language. A modeling tool should assist in the modeling work and not just in recording its result.

Also a method or an ad hoc way of working can benefit from the flexibility. For example, a method might state that the properties and classes of a subsystem are identified independently and they are combined in a separate step. Without any tool support the designer could first list the properties and the classes, group related properties together and then assign the groups to classes. The properties in a group become class properties of the class. With flexible modeling tool support, the designer could draw class properties without a class, visually group them and then move them into classes. If the tool is inflexible, the actual design work has to be done outside the tool, e.g. on pen and paper, and only the end result is entered into the tool.

At some point in the modeling, when the sketching is over, it is important that the model is and stays consistent with the modeling language. For instance, when the model has already gone through several iterations and only small incremental changes are being made, the changes must not break the model. To make full use of the model it should be complete and consistent when it is used, for example for generating code, simulating or testing. The usage dictates where in between permitting "everything" and full consistency the model consistency requirements should be set at a given time.

In addition to the modeling language itself, guidelines and profiles for the domain, organization or project may place restrictions on the model. Restrictions may affect the content, i.e. the model, and the appearance, i.e. the view data. For example, it may be required that classes have unique names within their namespace, or that in a workflow diagram the default path is laid out from left to right.

It is not possible or feasible to remove all restrictions on the models. A modeling tool can not help the user much if it can not make any assumptions about the model. The modeling tool interprets the user's actions in a certain way and thus excludes some options.

For example, when the user is drawing a line that represents an association and moves the end of the line over a class, the modeling tool assumes that the user intends to attach the association's end to the class. The modeling tool follows the UML class diagram metamodel and creates a link with the label *type* from the association's end to the class. However, no user action can connect these model elements with a link that is used between a class method and its parameter.

The interpretations that are coded into the integration component for a modeling tool impose restrictions, which in fact define a modeling language. This implied modeling language is a superset of the actual modeling language, i.e. it is more permissive. In Trinity it is considered, that there is an abstract metamodel, so called *relaxed metamodel* for the implied modeling language. The re-

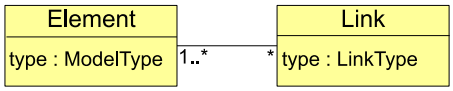

Fig. 4: The model repository's view of relaxed metamodel

laxed metamodel is specific to a modeling language, but also to a modeling tool and its integration component.

Each model of that language must comply with the restrictions defined by the relaxed metamodel. No collection of restrictions, including the actual metamodel of the language, can be looser than the relaxed metamodel.

From a purely implementation point of view, the model repository only requires that models consist of elements and links, both of which are typed. Links can have one or more ends, each of which is connected to an element. There are no restrictions on the types, so a class diagram may contain elements from a mind map and a link that should be between a state and a transition may be placed between two classes. These loose restrictions form a very permissive metamodel (Fig. 4), which is used for all the models regardless of the modeling language.

4.3 Setup for the Study

The study was carried out by the main researcher from the Trinity project and the author of DReAMT, who worked in the MoDES project. At the time of this writing, the study is still continuing and the work reported here was carried out in about a month's time in the beginning of 2010.

The goal was to build a model transformation to support the process of producing a model consistency rule set based on the metamodel and the relaxed metamodel of a language. The rule set creation process was at this stage restricted to UML class diagrams viewed and modified in Microsoft Visio. It was planned, that later the process would be generalized to apply to a wider context.

There was no rule checker component and no rule checking language yet at this point. So, there was no suitable intermediate language, which could capture the rules. It was therefore decided that the model transformation would produce the rule set in a simple XML format that lists the conflicting conditions. It was understood, that this format would not be the final modeling language and that once the rule checker component was developed, the target modeling language for the model transformation would change radically.

The Trinity researcher acted as the Design Phase Expert and the author of DReAMT acted as the Transformation Architect and as the Transformation Programmer. The Trinity researcher knew in detail how the Visio integration component worked and what limitations there were to editing the class diagrams. He also had ideas of specific consistency rules that should be enforced. However, he could not immediately craft an explicit relaxed metamodel nor could he list

detailed reasoning based on the differences in the relaxed and normal metamodels. The Trinity researcher had basic knowledge of model transformations and very little knowledge of DReAMT.

The author of DReAMT knew DReAMT thoroughly and as a model transformation researcher had good knowledge of model transformations. He had very little knowledge about the Visio integration, the relaxed metamodel or required consistency rules. Both participants had good knowledge of modeling and specifically modeling with UML.

4.4 Building the Model Transformation

First the participants met a couple of times to agree on the scope of the study, its goals, etc. After that they met roughly once a week in a room with a whiteboard. The whiteboard was used heavily to draw correspondence examples, metamodel and model fragments and to write patterns and consistency rules. The whiteboard was photographed when a session ended and when the whiteboard got full. The Transformation Architect then translated the information in the photographs into the DReAMT process artefacts offline. One of the photographs is shown in Fig. 5.

The Design Phase Expert and the Transformation Architect focused on crafting the relaxed metamodel and listing correspondence examples, consistency rules and reactions to the violations of the rules. A correspondence example represents a fragment of the source model(s) and a fragment of the target model(s). In this case the source models are the normal and relaxed metamodel and the target model is the consistency rule model. The source model fragments express a class of conflicts between the metamodels. They were given as class diagram fragments. The consistency rules and the reactions were just written as text.

For instance, the correspondence example in Fig. 6 shows a case where the metamodel requires an instance of a type (Y in the figure) to be always be contained within an instance of another type (X), but the relaxed metamodel permits stand-alone instances. A concrete example is the metamodel requiring class properties to be placed within classes.

When a designer is creating the consistency rules, they need to consider many things. Although a conflict between the actual metamodel and the relaxed metamodel always means that a model can be inconsistent, it does not necessarily mean that a rule should be generated.

The rule checker component will need to react to the modeling tool user's actions very fast and without slowing down the computer. The designer may deem that checking for some particular conflict would likely happen too frequently or be too slow to be suitable for an interactive application. Other conflicts might be unimportant or so frequent that notifying the tool user would be annoying instead of being helpful. It could also be, that checking for one conflict can be combined with checks for other conflicts or in general needs to be handled in an exceptional way. For these reasons, the model transformation can not be automatic.

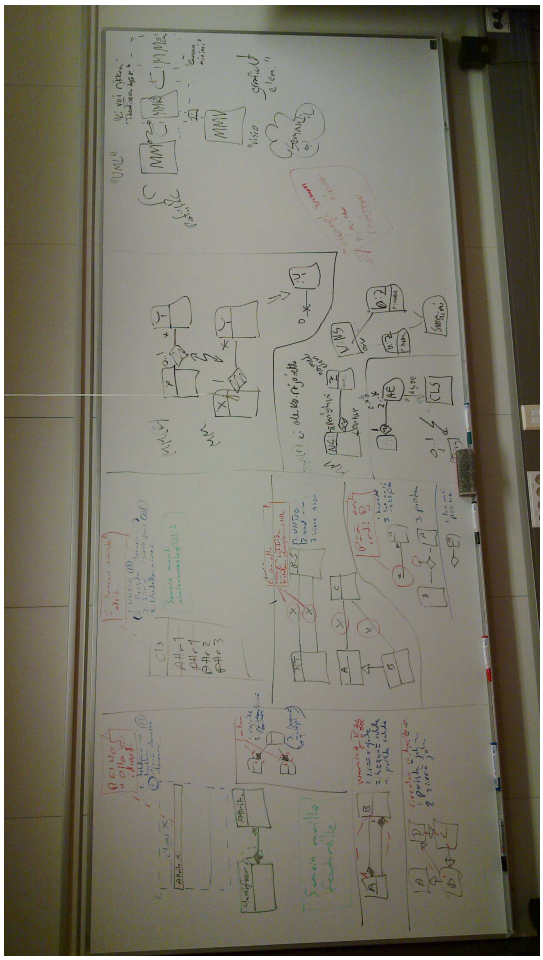

Fig. 5: Whiteboard full of correspondence examples, rules and fragments

The Design Phase Expert already knew before the transformation development process even started that some consistency rules were going to be needed. They were collected and they acted as a rough measure of how complete the transformational pattern system is, because all the rules should be found during the consistency rule creation process. If they are not found with the help of the patterns, then they need to be manually discovered by the designer.

Some correspondence examples were found by analyzing concrete conflicts between the normal and relaxed metamodel. Because there was no concrete relaxed metamodel, this step was not systematic. Correspondence examples were also found by looking at the list of consistency rules that should be found and trying to find a reasoning for them in the differences between the metamodels.

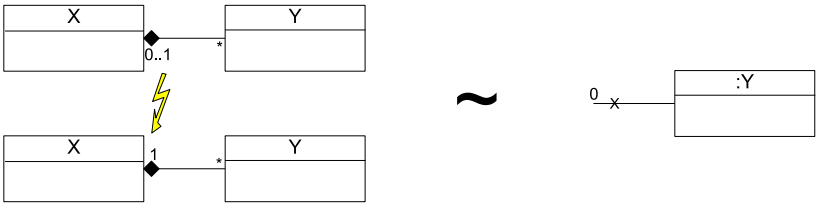

Fig. 6: Correspondence example for mandatory/optional container

In total about a dozen correspondence examples were found and refined into transformational patterns and implementation.

Because there was no explicit relaxed metamodel, there was nothing to execute the model transformation against. For this reason, the model transformation implementation step was skipped for the first few iterations. Instead, the transformational patterns were applied manually to selected parts of the metamodels by drawing the result on the whiteboard.

4.5 Observations and Conclusions

The iterations took roughly a week each, and both participants were busy with many other things at the same time. Having concrete artefacts in the form of correspondence examples and transformational patterns helped continue the work where it had been stopped the last time.

A fully automatic model transformation could not have been used in this case, because the designer's decision in two identical looking situations can be different. In fact, the designer needs to make quite many decisions. If the relaxed metamodel is changed, the validity of these decisions needs to be checked somehow. Decision context provided a convenient way to express validity conditions for decisions.

Because the transformational patterns were based on clearly specified conflicts between two metamodels, the decision contexts were easy to define and they were very small. For example, in the case of mandatory/optional containment, the conflict is the multiplicity of 1..1 in the metamodel and 0..1 in the relaxed metamodel. If the multiplicity on either metamodel changes, the association is no longer a containment association or the association is not between the same metaclasses, the decision becomes invalid, otherwise it is valid. This can be captured in a decision context.

5 Conclusions

In this report we discussed the application of DReAMT approach to model transformation development and the supporting tool. The tool was applied in two studies, one with Solita Oy and the other with a research project at TUT. The DReAMT approach was applied in the TUT research project.

The two studies reinforced our opinions that (i) the DReAMT approach can be used to develop model transformations; (ii) interactive model transformations are flexible; (iii) decision modeling makes automatic decision reuse possible; and (iv) the DReAMT tool can be used by people other than its author.

Acknowledgements

We would like to thank Solita Oy for the process modeling tool study and especially Antti Tirilä and Markus Kauko for their important roles in it. We would also like to thank Jari Peltonen for the flexible modeling tool study and for his effort as the Design Phase Expert. We owe thanks also to Mikko Hartikainen for developing the process modeling tool and the contained model transformation for the process modeling tool study.

References

1. Siikarla, M., Laitkorpi, M., Selonen, P., Systä, T.: Transformations have to be developed, ReST assured. In Vallecillo, A., Gray, J., Pierantonio, A., eds.: Theory and Practise of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 2008, Proceedings. Volume 5063 of Lecture Notes in Computer Science., Springer (July 2008) 1–15
2. Hammouda, I., Koskinen, J., Pussinen, M., Katara, M., Mikkonen, T.: Adaptable concern-based framework specialization in UML. In: Proceedings of ASE 2004, IEEE Computer Society (September 2004) 78–87
3. Eclipse: Model Development Tools (MDT). (2009) On-line at `http://www.eclipse.org/modeling/mdt/?project=uml2tools`.
4. Eclipse: Eclipse - an open development platform. (2007) On-line at `http://www.eclipse.org/`.
5. Siikarla, M., Systä, T.: Transformational pattern system - some assembly required. In Bruni, R., Varró, D., eds.: Proceedings of GT-VMT 2006. (April 2006) 57–68
6. Siikarla, M., Systä, T.: Decision reuse in an interactive model transformation. In: 12th European Conference on Software Maintenance and Reengineering, CSMR 2008, April 1-4, 2008, Athens, Greece, IEEE (April 2008) 123–132 On-line at `http://dx.doi.org/10.1109/CSMR.2008.4493307`.
7. Siikarla, M.: DReAMT: A tool set for interactive model transformations. In: Proceedings of the Nordic Workshop on Model Driven Engineering, NW-MoDE'08, Reykjavik, Iceland (August 2008) 1–15
8. OMG: MDA guide version 1.0.1 (2003) On-line at `http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf`.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FIN-33101 Tampere, Finland