Holger Giese, Andy Schürr, Albert Zündorf (Eds.)

FUJABA Days 2004

15th -17th September 2004
TU Darmstadt, Germany

Proceedings

Volume Editors

Jun.-Prof. Dr. Holger Giese
University of Paderborn
Department of Computer Science
Warburger Straße 100, 33098 Paderborn, Germany
hg@uni-paderborn.de

Prof. Dr. Andy Schürr
TU Darmstadt
Department of Electrical Engineering and Information Technology
Merckstr. 25, 64283 Darmstadt, Germany
andy.schuerr@es.tu-darmstadt.de

Prof. Dr. Albert Zündorf
University of Kassel
Department of Computer Science and Electrical Engineering
Wilhelmshöher Allee 73, 34121 Kassel, Germany
Albert.Zuendorf@uni-kassel.de

Program Committee

Program Committee Chairs

Andy Schürr (TU Darmstadt, Germany)
Albert Zündorf (University of Kassel, Germany)

Program Commitee Members

Bernhard Rumpe (TU Braunschweig, Germany)
Holger Giese (University of Paderborn, Germany)
Jürgen Börstler (University of Umea, Sweden)
Jens Jahnke (University of Victoria, Canada)
Luuk Groenewegen (Leiden University, Netherlands)
Manfred Nagl (RWTH Aachen, Germany)
Pieter Van Gorp (University of Antwerp, Belgium)
Tarja Systä (Tampere University of Technology, Finland)
Wilhelm Schäfer (University of Paderborn, Germany)

Editors' preface

Fujaba is an Open Source model transformation tool which combines features of commercial "Executable UML" CASE tools with rule-based visual programming concepts adopted from its ancestor, the graph transformation tool PROGRES. The Fujaba project started at the software engineering group of Paderborn University in 1997. In 2002 Fujaba has been redesigned and became the Fujaba Tool Suite with a plug-in architecture allowing developers to add functionality easily while retaining full control over their contributions. This new architecture simplified the cooperation of the research groups at different Universities, which form nowadays the core of the Fujaba Development Project.

At the early days, Fujaba had a special focus on code generation from UML diagrams resulting in a visual programming language with a special emphasis on object structure manipulating rules. Today, at least six rather independent tool versions are under development in Paderborn, Kassel, and Darmstadt for supporting (1) reengineering, (2) embedded real-time systems, (3) education, (4) specification of distributed control systems, (5) integration with the ECLIPSE platform, and (6) MOF-based integration of system (re-)engineering tools.

According to our knowledge, quite a number of additional research groups all over the world have also chosen Fujaba as a platform for UML related research activities. Therefore, the 2nd International Fujaba Days had the aim to bring together Fujaba developers and Fujaba users from all over the world to present their ideas and projects and to discuss them with each other and with the Fujaba core development team.

More specifically, thanks to the EU research project SEGRAVIS we were able to invite Birgit Demuth from Dresden University and Pierre Alain Muller from the University of Rennes to present the Dresden OCL Toolkit and the TOPModl initiative for the development of an European Open Source Model Transformation toolkit. Both talks provided us with the needed input for the discussion of possibilities of cooperation and combination of Fujaba, the Dresden OCL Toolkit and the TOPModl toolkit. Beside these two invited talks, 14 groups of authors attended the workshop and presented their ideas how to use, modify, and extend Fujaba. In addition, one day of the workshop was reserved for joint programming activities.

The abstracts of the two invited talks plus the extended abstracts of the 14 accepted submissions to the workshop are compiled in the form of a technical report. The structure of the report reflects the organization of the related presentations in sessions at the workshop. There were two sessions related to the topic of improving the FUJABA infrastructure, two sessions presenting an impressive variety of applications developed with Fujaba as well as one session dealing with the development of new Fujaba plugins.

We hope that this compilation of the abstracts of the Fujaba workshop presentations gives you (the reader) an impression about ongoing activities in the Fujaba project and provides you with the needed background information and motivation for joining the Fujaba development team.

The Editors (Andy, Albert, and Holger)

Table of Contents

Invited Talk

Structure of the Dresden OCL Toolkit 1
Birgit Demuth et al. (TU Dresden)

Improving the FUJABA Infrastructure I

Towards Incremental Graph Transformation in Fujaba *3*
Gergely Varró (Budapest University of Technology and Economics)
Selective Tracing of Java Programs *7*
Lothar Wendehals et al. (University of Paderborn)

Application Development with FUJABA

Fujaba-based Tool Development and Generic Activity Mapping:
Building an eHomeConfigurator *11*
Ulrich Norbisrath et al. (RWTH Aachen)
Simulating and Testing of Mobile Computing Systems Using Fujaba *15*
Ping Guo et al. (University of Paderborn)
Design and Simulation of Self-Optimizing Mechatronic Systems with
Fujaba and CAMeL *19*
Sven Burmester et al. (University of Paderborn)

New Plugins for the FUJABA Community

Modifications of the FUJABA Statechart Interpreter for Multiagent-based
Discrete Event Simulation *23*
Nicolas Knaak (University of Hamburg)
Component Templates for Dependable Real-Time Systems *27*
Matthias Tichy et al. (University of Paderborn)
Visualizing Differences of UML Diagrams with Fujaba *31*
Jörg Niere (Uni Siegen)

(Meta) Model-Driven Application Development with FUJABA

An Adaptable TGG Interpreter for In-Memory Model Transformations *35*
Robert Wagner et al. (University of Paderborn)
Standardizing SDM for Model Transformation *39*
Pieter Van Gorp et al. (Universiteit Antwerpen)
A MOF 2.0 Editor as Plug-in for FUJABA *43*
Carsten Amelunxen (TU Darmstadt)

Invited Talk

The TOPModl Initiative *49*
Pierre Alain Muller et al. (Université de Rennes)

Improving the FUJABA Infrastructure II

Adding Pluggable Meta Models to FUJABA *57*
Tobias Rötschke (TU Darmstadt)
FASEL: scripted Backtracking in FUJABA *63*
Boris Böhlen et al. (RWTH Aachen)
Yet Another Association Implementation *67*
Thomas Maier et al. (University of Kassel)

Structure of the Dresden OCL Toolkit

Extended Abstract

Birgit Demuth
Dresden University of Technology
Department of Computer Science
Dresden, Germany
Birgit.Demuth@inf.tu-dresden.de

Sten Loecher
Dresden University of Technology
Department of Computer Science
Dresden, Germany
Sten.Loecher@inf.tu-dresden.de

Steffen Zschaler
Dresden University of Technology
Department of Computer Science
Dresden, Germany
Steffen.Zschaler@inf.tu-dresden.de

The Object Constraint Language (OCL) as a part of the UML standard [1] is a formal language for defining constraints on UML models. We present a software platform for OCL tool support [2]. The platform is designed for openness and modularity, and is provided as open source. The goal of this platform is, for one thing, to enable practical experiments with various variants of OCL tool support, and then, to allow UML tool builders and users to integrate and adapt the existing OCL tools into their own environments. The Dresden OCL Toolkit provides the following tools:

OCLCore: The base tool of the OCL toolkit consists of four different modules:

- The **OCLParser** transforms the input OCL expression into an abstract syntax tree [3]. The abstract syntax tree forms the common data representation for all other tools in the toolkit.
- The **OCLEditor** is a comfortable editor which includes, besides editing of constraints, features like a toolbar and adequate error messages. The user interface is designed to allow the integration of the OCL editor not into a specific UML tool, but into various environments. The screenshot in Figure 1 gives an impression of the OCL editor integrated into Together. It shows on the right hand side a UML class diagram for a simple hotel reservation system. On the left hand side, an OCL constraint has been added asserting that the region of a hotel must be the same as the region of the hotel's destination.
- The **OCLTypeChecker** checks type correctness of OCL expressions and offers type information towards other modules. Necessary UML model information has to be extracted from the environment in which the OCL toolkit is embedded. For this purpose a small external interface (called **ModelFacade**) is provided [3].
- The **OCLNormaliser** transforms the abstract syntax tree into a *normal form* of OCL terms, such that all terms can be mapped into a subset of the OCL language more adequate for subsequent tasks. That way it can be avoided that every tool using OCLCore has to implement the execution of every OCL expression completely.

OCL2Java: This tool transforms a normalised syntax tree into Java Code. It uses a class library which offers Java representations for the predefined OCL types.

OCLInjector4Java: The tool takes the generated Java code and inserts it into an application program. This code instrumentation is done by the generation of wrapper methods for all methods whose compliance to specified OCL constraints is to be checked during execution. The used technique including code cleaning is described in [6]. OCLInjector4Java has been integrated into ArgoUML and Together.

OCL2SQL: The SQL code generator [4] generates an SQL check constraint, assertion or trigger for an OCL invariant based on the accordingly normalised abstract syntax tree. OCL2SQL can be used and adapted for different relational database systems and different object-to-table mappings. Similarly to OCLTypeChecker's ModelFacade, we provide an interface for the integration of various strategies of object-to-table mapping.

OCLInterpreter: A first tool developed outside of the Dresden University of Technology is an OCL interpreter that allows the dynamic checking of OCL constraints against objects. The OCLInterpreter is also designed based on a normalised abstract syntax tree.

OCL20: All previous tools establish an architecture which is designed for OCL 1.x support. Currently we are reengineering the Dresden OCL Toolkit according to the new requirements of the revised and approved specification of OCL ("OCL 2.0" [7]). The OCL20 module is a prototype of a metamodel-based OCL compiler consisting of a MOF repository implementation and a code generator [5]. The OCL 2.0 parser is still under development. The research issue is to which extent a parser can be automatically generated from the provided specification.

An important requirement on tools supporting OCL is their cooperation with UML tools. The specification of OCL con-

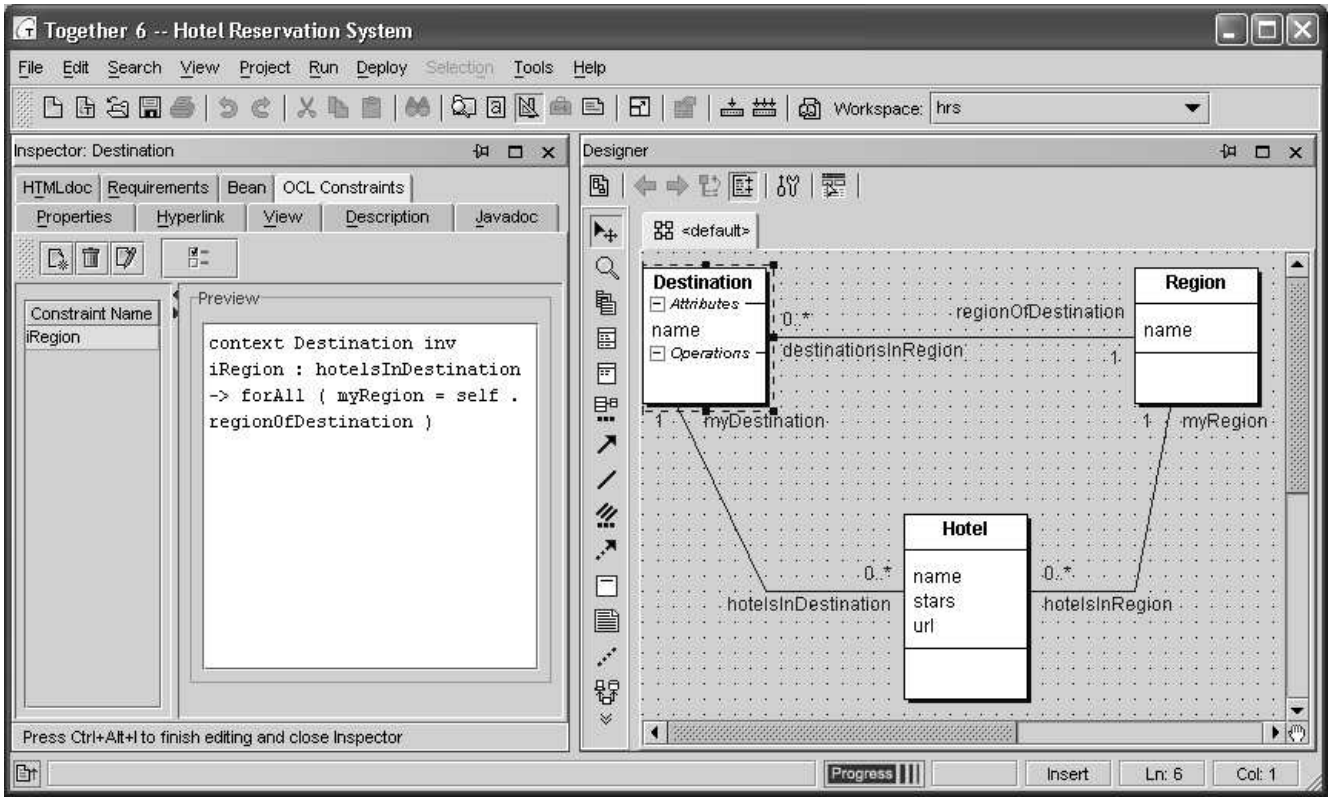

Figure 1: Dresden OCL Toolkit integrated into Together

straints without any model makes no sense. OCLTypeChecker's ModelFacade provides support for this flexibility. We have implemented the ModelFacade in the following ways: An OCL tool can be **tightly** integrated into a UML tool as an add-in. Then the model interface must be implemented by an integration component accessing the UML tool's repository. Examples for this technique are the integration of our toolkit into Together (see Figure 1), ArgoUML, Poseidon, and Rational Rose. A kind of **loose** integration is the use of XMI files for static UML model information. The Dresden OCL toolkit already provides the necessary ModelFacade implementation to use this technology.

1. REFERENCES

[1] OMG UML v. 1.5 specification, www.omg.org/technology/documents/formal/uml.htm

[2] Dresden OCL Toolkit, http://dresden-ocl.sourceforge.net/

[3] Hussmann, H., Demuth, B., Finger, F.: Modular Architecture for a Toolset Supporting OCL. in: Third Int. Conference on the Unified Modeling Language (UML'2000), York, UK, October 2000, Springer, 2000

[4] Demuth, B., Hussmann, H., Loecher, St.: OCL as a Specification Language for Business Rules in Database Applications. in: Fourth Int. Conference on the Unified Modeling Language (UML 2001), Toronto, Canada, October 1-5, 2001

[5] Loecher, St., Ocke, St.: A Metamodel-Based OCL-Compiler for UML and MOF. in: Workshop OCL 2.0 - Industry standard or scientific playground?, Sixth Int. Conference on the Unified Modelling Language (UML 2003), October 21, 2003, San Francisco, i11www.ilkd.uni-karlsruhe.de/~baar/oclworkshopUml03

[6] Wiebicke, R., Utility Support for Checking OCL Business Rules in Java Programs. Masters Thesis, Dresden University of Technology, 2000, dresden-ocl.sourceforge.net/

[7] OCL 2.0 Submission, www.klasse.nl/ocl/ocl-subm.html

Towards Incremental Graph Transformation in Fujaba

[Position paper]

Gergely Varró
Department of Computer Science and Information Theory
Budapest University of Technology and Economics
Magyar tudósok körútja 2.
H-1521 Budapest, Hungary
gervarro@cs.bme.hu

ABSTRACT

I discuss a technique for on-the-fly model transformations based on *incremental updates*. The essence of the technique is to keep track of all possible matchings of graph transformation rules, and update these matchings incrementally to exploit the fact that rules typically perform only local modifications to models. The proposal is planned to be implemented as a plug-in for the Fujaba graph transformation framework.

Keywords

graph transformation, graph pattern matching, incremental updates, Fujaba

1. INTRODUCTION

Model Driven Architecture. Recently, the Model Driven Architecture (MDA) of the Object Management Group (OMG) has become an interesting trend in software engineering. The main idea of the MDA framework is the use of models during the entire system design cycle. A major factor in the success of MDA is the development of industrial-strength models and various modeling languages. Several metamodeling approaches [2, 6, 19] have been developed to provide solid foundations for language engineering to allow system engineers to design a language for their own domain. As being the standard and visual object-oriented modeling language, UML obviously plays a key role in language design.

Transformation engineering in MDA. [20] However, the role of model transformations between modeling languages within MDA is as critical as the role of modeling languages themselves. As model transformations required by the MDA framework are supposed to be mainly developed by software engineers, precise yet intuitive notations are required for model transformation languages. QVT [16], an initiative of the OMG, aims at developing a standard for capturing Queries, Views and Transformations in MDA.

Incremental model transformations. During the design phase of the software engineering process, the system model may be modified several times, e.g., when correcting bugs, performing refinement steps, etc. When only a small portion of the model is modified, it is enough in general to re-execute a model transformation only on the part of the model that has actually been changed. This approach is called an *incremental (or on-the-fly) model transformation*.

The most typical example in a UML context is the incremental update of various views. A UML diagram shows one aspect of the system under design. If the system engineer modifies only one diagram, then modification may result in an inconsistent model. In order to maintain consistency, the design process should be supported by incremental model transformation, which updates all UML diagrams in a consistent way whenever any diagram changed. A related topic is discussed in [12], where consistency of logical and conceptual schemata of databases is maintained incrementally using traditional graph transformation techniques.

Incremental model transformations would also be advantageous for visual modeling languages. For instance, in [3], the authors discuss how the concrete syntax of a language can be generated from the abstract syntax by batch model transformations. However, incremental transformations would make this technique eligible to visual language editors, which require to automatically update the concrete syntax of the model according to the model-view-controller paradigm.

Fujaba as a model transformation tool. Fujaba, which is an Open Source UML CASE tool provides a rule-based visual programming language for manipulating the object structure based on the paradigm of graph transformation [18].

Traditionally, Fujaba has supported the specification of (and code generation from) the dynamic behavior of the system in the form of UML activity diagrams. Activity diagrams define the control flow of the methods and as such, they consist of activities (nodes) and transitions (edges). The role of transitions is to define temporal dependencies (i.e., execution order) between activities.

A graph transformation rule describes the behavior of a specific activity. A simplified version of UML collaboration diagrams (referred as story patterns) is used for specifying graph transformation rules. Activity diagrams that contain story patterns as activities are called *story diagrams* [8]. However, while Fujaba is considered to be one of the fastest graph transformation engines, there is still lack of support for incremental transformations.

Fujaba has been redesigned, and currently, it has a plug-in architecture. This new architecture still supports the basic code generation feature, but it additionally allows developers to easily add different functionalities while retaining full control over their contributions. As a consequence of this flexibility, several application areas exist such as re-engineering [14], embedded real-time system design [1], education [15], etc.

Objectives. In the paper, I discuss the concepts of on-the-fly model transformation based on *incremental updates*. The essence of the technique is to keep track of all possible matchings of graph transformation rules, and update these matchings incrementally to exploit the fact that rules typically perform only local modifications to models. I plan to implement such an incremental graph transformation engine using Rete-algorithms [9]. The engine is planned to be integrated into the Fujaba graph transformation framework as a plug-in.

2. MODEL TRANSFORMATION

Visual modeling languages are frequently described by a combination of metamodeling and graph transformation techniques [6, 19].

2.1 Metamodeling

The *metamodel* describes the abstract syntax of a modeling language. Formally, it can be represented by a type graph. Nodes of the type graph are called *classes*. A class may have *attributes* that define some kind of properties of the specific class. *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra attributes. Finally, *associations* define connections between classes.

In the MOF terminology [17], a metamodel is defined visually in a UML class diagram notation. In practical terms, the class diagram that has been designed in Fujaba by system engineers will form the metamodel in this case.

The *instance model* (or, formally, an instance graph) describes concrete systems defined in a modeling language and it is a well-formed instance of the metamodel. Nodes and edges are called *objects* and *links*, respectively. Objects and links are the instances of metamodel level classes and associations, respectively. Attributes in the metamodel appear as *slots* in the instance model. Inheritance in the instance model imposes that instances of the subclass can be used in every situation, where instances of the superclass are required. In case of Fujaba, the generated concrete system will form the instance model.

Example. A distributed mutual exclusion algorithm whose full specification can be found in [11] will serve as a running example throughout the paper. *Processes* try to access shared *resources* in this domain. One requirement from the algorithm is to allow access to each resource by at most one process at a time. This is fulfilled by using a token ring, which consists of processes connected by edges of type *next*. In the consecutive phases of the algorithm, a process may issue a *request* on a resource, the resource may eventually be *held by* a process and finally a process may *release* the resource. The right to access a resource is modeled by a *token*. The algorithm also contains a deadlock detection procedure, which has to track the processes that are *blocked*.

The metamodel (type graph) of the problem domain and a sample instance model are depicted in the left and right parts of Fig. 1, respectively. The instance model presents a situation with two processes that are linked to each other by edges of type *next*.

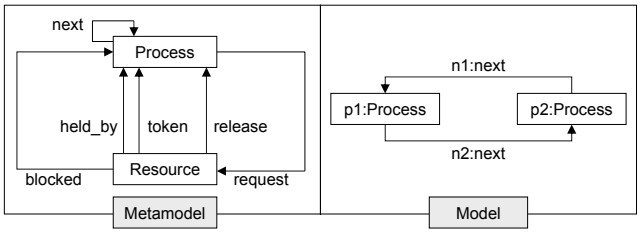

Figure 1: A sample metamodel and instance model

2.2 Graph transformation

Graph transformation [5, 18] provides a pattern and rule based manipulation of graph-based models. Each rule application transforms a graph by replacing a part of it by another graph.

A *graph transformation rule* $\mathsf{r} = (\mathsf{LHS}, \mathsf{RHS}, \mathsf{NAC})$ contains a left–hand side graph LHS, a right–hand side graph RHS, and negative application condition graphs NAC.

The *application* of r to an *host (instance) model* M replaces a matching of the LHS in M by an image of the RHS. This is performed by (i) finding a matching of LHS in M (by graph pattern matching), (ii) checking the negative application conditions NAC (which prohibit the presence of certain objects and links) (iii) removing a part of the model M that can be mapped to LHS but not to RHS yielding the context model, and (iv) gluing the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) obtaining the *derived model* M'. The latter two steps form the so-called updating phase. A *graph transformation* is a sequence of rule applications from an initial model M_I.

Example. A sample rule of the distributed mutual exclusion algorithm (depicted in Fig. 2) simply inserts a new process between neighboring processes p1 and p2.

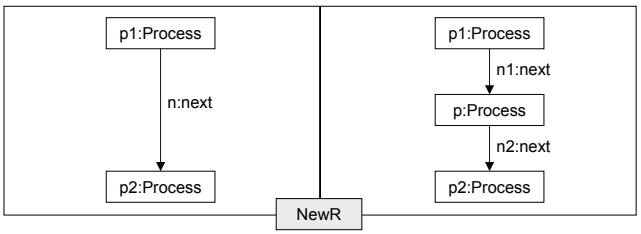

Figure 2: A sample transformation rule (newR)

2.3 Graph pattern matching

Typically, the most critical phase of a graph transformation step concerning the overall performance is graph pattern matching, i.e. to find a single (or all) occurrence(s) of a given LHS graph in a host model.

Current graph transformation engines use different sophisticated strategies in the graph pattern matching phase. These strategies can be grouped into two main categories.

- Algorithms based on *constraint satisfaction* (such as [13] in AGG [7], VIATRA [21]) interpret the graph elements of the pattern to be found as variables which should be instantiated by fulfilling the constraints imposed by the elements of the instance model.
- Algorithms based on *local searches* start from matching a single node and extending the matching to the neighboring nodes and edges. The graph pattern matching algorithm of PROGRES (with search plans [23]), Dörr's approach [4], and the object-oriented solution in FUJABA [8] fall in this category.

However, it is common in all these engines that they can be characterized as having a complex pattern matching phase followed by a simple modification phase and these phases are executed iteratively.

The main problem is that the information on previous match is lost, when a new rule application is started. As a consequence, the complex pattern matching phase has to be executed from scratch again and again. However, because of the local nature of modifications, it may be expected that the majority of matchings remain valid in consecutive steps. The same matchings are calculated several times, which seems to be a waste of resources in case of e.g., long transformation sequences.

3. INCREMENTAL UPDATES

In order to avoid recalculation of matchings, we proposed a technique based on *incremental updates* [22], for implementing efficient graph transformation engines designed especially for incremental (on-the-fly) model transformations. The basic idea in a graph transformation context is to store information on previous match and to keep track of modifications.

Several other solutions already exist for reducing the overhead of finding matches for LHS of rules as implemented in PROGRES [23]: (i) applying a graph transformation to all matches in the graph as one graph rewriting step (pseudo-parallel graph transformation), (ii) using incrementally computed derived attributes and relationships in LHS, and (iii) using rule parameters in graph transformations to pass computed knowledge about possible LHS matches from one rule to the next one.

After many years of research, different techniques based on the incremental updating idea have evolved and by now they are widely accepted and successfully used in several types of applications (e.g., relational databases, expert systems).

- In the area of relational databases, views may be updated incrementally. A database view is a query on a database that computes a relation whose value is not stored explicitly in the database, but it appears to the users of the database as if it were. However, in a group of methods, which is called by view materialization approach, the view is explicitly maintained as a stored relation [10]. Every time a base relation changes, the views that depend on it may need to be recomputed.

- In the area of rule-based expert systems, the Rete-algorithm (for more details see [9]) uses the idea of incremental pattern matching for facts. First a data-flow network is constructed based on the condition (*if*) parts of rules, which is basically a directed acyclic graph of a special structure. Initially, this network is fed by basic facts through its input channels. Compound facts are constituted of more elementary facts, thus they are the inputs of internal nodes in the network. If a fact reaches a terminal node, then the rule related to this specific node becomes applicable and assignments modifying the set of basic facts may be executed (according to the *then* part). Since every node keeps a record of its input facts, only modifications of these facts have to be tracked at each step.

Despite these results, (quite surprisingly) no graph transformation tools exist that provide support for incremental transformations. In [22], we carried out some initial experiments, which used an off-the-shelf relational database to measure the performance of the incremental updating method compared to the traditional (from scratch) approach. However, it turned out the most relational databases do not support incremental view updates. Therefore, it seems to be necessary to develop a new incremental graph transformation engine from scratch.

In the current paper, I propose to build a graph transformation engine that uses the Rete-algorithm for implementing the incremental updating technique.

Now I sketch the basic structure of such an engine. A graph transformation rule can be viewed as a rule that has a condition (*if*) and an action (*then*) part. The condition part corresponds to the LHS of the graph transformation rule, while the action part consists of all the actions (delete, update, insert) that have to be executed in the updating phase. According to this mapping, we can build a data-flow network for each rule using the LHS. Nodes and edges of the LHS are mapped to input nodes, while the whole LHS will correspond to a terminal node. The data-flow network may also have some internal nodes, which are basically subgraphs of the LHS. After this network building phase we will have as many data-flow (Rete) networks as many rules we originally have. Then these networks are merged by the Rete-algorithm in order to decrease the number of nodes.

Note that the nodes and edges of the metamodel and the actual instance model will appear as input nodes and basic facts assigned to the corresponding input nodes, respectively. Basic facts flow through the network and constitute more and more compound facts as they progress. When a compound fact reaches a terminal node, then the corresponding graph transformation rule becomes applicable, and the updating phase can be executed. This phase actually modifies the active set of basic facts assigned to input nodes.

In an ideal case, such an incremental graph transformation engine should be available as a plug-in for many graph transformation tools (thus being independent of them). However, since the interfaces of graph transformation tools are not (yet) standardized I plan to integrate the incremental engine as a transformation plug-in of Fujaba. This would provide an *alternate graph transformation engine* tailored especially to incremental model transformations (possibly defined by triple graph grammar rules). However, no modifications are required to the base system of Fujaba.

Example. In order to sketch the idea of incremental updates, let us consider that rule newR (depicted in Fig. 2) is trying to be applied to the instance model of Fig. 1. The pattern matching phase selects two valid subgraphs of the instance model, on which the rule is applicable. The transformation engine then executes the updating phase resulting in a model that contains 3 processes that are stringed on a chain consisting of 3 edges of type *next*.

Up to this point, both traditional and incremental approaches do the same. But when the pattern matching phase of the following rule application is executed, the traditional approach recalculates valid matchings from scratch, while the incremental method only has to delete invalid matchings and generate new ones. The first method should examine all the *next* edges appearing in the instance model, which may contain an arbitrary number of *next* edges. However, in case of the incremental technique, it is enough to examine only such *next* edges that are actually removed or created in the previous step. The number of such edges are always three in this example regardless of the size of the instance model.

Naturally, in case of dozens (hundreds) of transformation rules, a single application of a rule might need to recalculate the matching of several rules therefore, there is certainly a trade-off between a cheap pattern matching phase and a more complex update phase. I also intend to carry out experiments to assess this trade-off between traditional (batch or programmed) and incremental transformations.

4. CONCLUSIONS

In this paper, I discussed the necessity of incremental model transformations in the context of the Model Driven Architecture (transformation-based derivation of concrete syntax from abstract syntax in visual modeling languages, consistent and on-the-fly update of UML diagrams, etc.). I discussed the concepts of incremental model transformations based on the paradigm of graph transformation. I plan to implement such an engine using Rete-algorithms and integrate it into Fujaba as a plug-in. Furthermore, I would like to investigate the applicability of the incremental approach to various model transformation techniques (including triple graph grammars).

5. ACKNOWLEDGMENT

I am very much grateful to Dániel Varró and Andy Schürr for giving valuable comments and hints on incremental updates strategies and/or the paper itself.

6. REFERENCES

[1] S. Burmester and H. Giese. The Fujaba real-time statechart plugin. In *Proc. of the Fujaba Days 2003, Kassel, Germany*, October 2003.

[2] T. Clark, A. Evans, and S. Kent. The Metamodelling Language Calculus: Foundation semantics for UML. In H. Hussmann, editor, *Proc. Fundamental Approaches to Software Engineering, FASE 2001 Genova, Italy*, volume 2029 of *LNCS*, pages 17–31. Springer, 2001.

[3] P. Domokos and D. Varró. An open visualization framework for metamodel-based modeling languages. In *Proc. GraBaTs 2002, International Workshop on Graph-Based Tools*, volume 72 of *ENTCS*, pages 78–87, Barcelona, Spain, October 7–8 2002. Elsevier.

[4] H. Dörr. *Efficient Graph Rewriting and Its Implementation*, volume 922 of *LNCS*. Springer-Verlag, 1995.

[5] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications, Languages and Tools*. World Scientific, 1999.

[6] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.

[7] C. Ermel, M. Rudolf, and G. Taentzer. *In [5]*, chapter The AGG-Approach: Language and Tool Environment, pages 551–603. World Scientific, 1999.

[8] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language. In G. R. G. Engels, editor, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, volume 1764 of *LNCS*. Springer Verlag, 1998.

[9] C. L. Forgy. RETE: A fast algorithm for the many pattern/many object match problem. *Artificial Intelligence*, 1982.

[10] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 1995.

[11] R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering: First International Conference, FASE'98*, volume 1382 of *LNCS*, pages 138–153. Springer-Verlag, 1998.

[12] J. H. Jahnke, W. Schäfer, J. P. Wadsack, and A. Zündorf. Supporting iterations in exploratory database reengineering processes. *Science of Computer Programming*, 45(2-3):99–136, 2002.

[13] J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12(4):403–422, 2002.

[14] J. Niere. Using learning toward automatic reengineering. In *Proc. of the 2nd International Workshop on Living with Inconsistency*, 2001.

[15] J. Niere and C. Schulte. Thinking in object structures: Teaching modelling in secondary schools. In *Proceedings of the ECOOP Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts*, 2002.

[16] Object Management Group. *QVT: Request for Proposal for Queries, Views and Transformations*.

[17] Object Management Group. *Meta Object Facility Version 2.0*, April 2003.

[18] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. World Scientific, 1997.

[19] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.

[20] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *Proc. UML 2004: 7th International Conference on the Unified Modeling Language*, 2004. In press.

[21] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002.

[22] G. Varró and D. Varró. Graph transformation with incremental updates. In *Proc. 4th Int. Workshop on Graph Transformation and Visual Modeling Techniques*, 2004.

[23] A. Zündorf. Graph pattern-matching in PROGRES. In *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *LNCS*, pages 454–468. Springer-Verlag, 1996.

Selective Tracing of Java Programs*

Lothar Wendehals, Matthias Meyer, Andreas Elsner
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
[lowende|mm|trinet]@upb.de

Abstract

Design recovery, which means extracting design documents from source code, is usually done by static analysis techniques. Analysing behaviour by static analysis is very imprecise. Therefore, we combine static and dynamic analysis to increase the preciseness of our design recovery process.

In this paper we present an approach to collect data for the dynamic analysis by recording method calls during a program's execution. To reduce the amount of information we monitor only relevant classes and methods identified by static analysis. We developed a new plug-in for the Fujaba Tool Suite *called* JavaTracer *which we use for the recording of method calls in Java programs.*

1. Motivation

Today software engineers spend most of their time maintaining software systems. The documentation of such systems is often not available or has become obsolete. Before a system can be changed to meet new requirements it has to be reverse engineered and its design has to be recovered which is a time consuming and expensive task.

We developed a tool-supported semiautomatic approach to design recovery [4] within the Fujaba Tool Suite [6]. The approach facilitates the recognition of patterns such as design patterns [1] in the source code of a system. It is a highly scaleable process which can be applied to large real world applications.

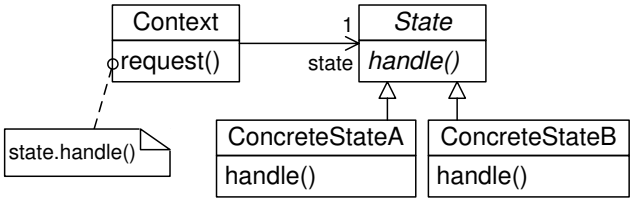

Figure 1: The *State* design pattern

So far we only perform a static analysis based on source code that focuses mainly on the structural aspects of a pattern. However, many patterns are structurally very similar and differ only in their behaviour, e.g. the design patterns *State* (cf. Figure 1) and *Strategy* [8, 1]. Those behavioural differences can only be recognized during a dynamic analysis of the system. Therefore, we will combine our static analysis with a subsequent dynamic analysis [7, 8].

As the basis for dynamic analysis a program trace will be recorded during the execution of the program to be analysed. Since the amount of information for a complete program trace is too high, we record only relevant method traces. The relevant classes and methods to be monitored are identified by the static analysis.

In the next section we present an overview of our design recovery process. The selective recording of program traces is described in Section 3. Related work follows in Section 4. In Section 5 we report about the performance of our approach. A short summary of future work follows in Section 6.

2. The Design Recovery Process

Our design recovery process is based on an extended Abstract Syntax Graph (ASG) representation of the source code. The ASG includes method bodies for a rudimentary static analysis of behaviour. During design recovery the ASG will be annotated by nodes which are linked to an arbitrary number of ASG nodes to mark recognized pattern instances.

A tool-based pattern recovery requires a formal definition of patterns. Thus, for each pattern to be recognized within the source code a structural and a behavioural pattern is given. The process starts with the static analysis using the structural patterns. During this phase pattern instance candidates are recognized. These candidates will be verified by the subsequent dynamic analysis using the behavioural patterns.

2.1 Static Analysis

The structural patterns are specified as graph grammar rules with respect to the ASG [4]. Graph grammar rules consist of a left-hand side (LHS) and a right-hand side (RHS). The LHS describes a sub graph to be found within the host graph. The RHS describes the modifications of the sub graph when the rule is applied.

Figure 2 depicts a structural pattern for the *State* design pattern. The LHS and RHS of the graph grammar rule are defined by one graph. The LHS is defined by all black nodes and edges and describes the sub graph to be found within the ASG. The RHS consists of the LHS and additional nodes and edges marked with the stereotype ≪create≫. It describes how to mark the found sub graph by creating an annotation node and links to ASG nodes.

The *State* pattern (cf. Figure 1) enables an object to change its behaviour at runtime by changing its internal

*This work is part of the Finite project funded by the German Research Foundation (DFG), project-no. SCHA 745/2-1.

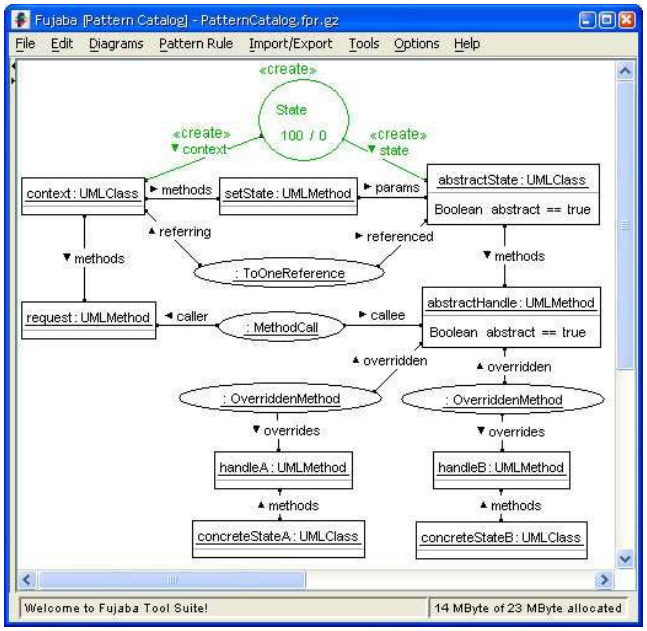

Figure 2: Structural pattern for *State*

state [1]. Each state is represented by a separate class which encapsulates the state-specific behaviour. The state classes adhere to a common interface defined by an abstract super class. The object references exactly one state object and delegates requests to this state object.

This structure is described by the LHS in Figure 2. It specifies that the ASG must contain a class context:Class which references an abstract class abstractState:Class. This is expressed by the oval annotation node of type ToOneReference. Note, that the LHS may also contain annotation nodes created by the application of other rules. This enables the composition of structural patterns.

In addition, the class context is required to have a method setState:Method which has a parameter of type abstractState:Class and another method request:Method which calls (MethodCall [1]) an abstract method abstractHandle:Method of class abstractState:Class. Furthermore, the abstract method abstractHandle:Method has to be overridden by at least two concrete methods (handleA:Method and handleB:Method) in two subclasses of class abstractState:Class, namely concreteStateA:Class and concreteStateB:Class.

If the rule can be applied, i.e. the sub graph can be found, it creates a State annotation node and links it to the context:Class and abstractState:Class classes. The mapping between nodes of the LHS and the found sub graph nodes is stored for dynamic analysis.

The application of the graph grammar rules for the structural patterns recovers pattern instance candidates. For details on the rule application cf. [4]. The specification of the structural patterns and their recognition are implemented by the FUJABA plug-ins PATTERNSPECIFICATION and INFERENCEENGINE, respectively. The JAVAPARSER plug-in is used to generate an ASG representation of Java source code thereby allowing the analysis of Java programs. Note, however, that the approach is not limited to Java.

[1]Polymorphism and dynamic method binding prevent a precise static analysis of method calls.

2.2 Dynamic Analysis

The purpose of the dynamic analysis is to verify the pattern instance candidates recognized by the preceding static analysis. It has to be checked if the collaboration of the candidate's classes during runtime matches the pattern's behavioural description.

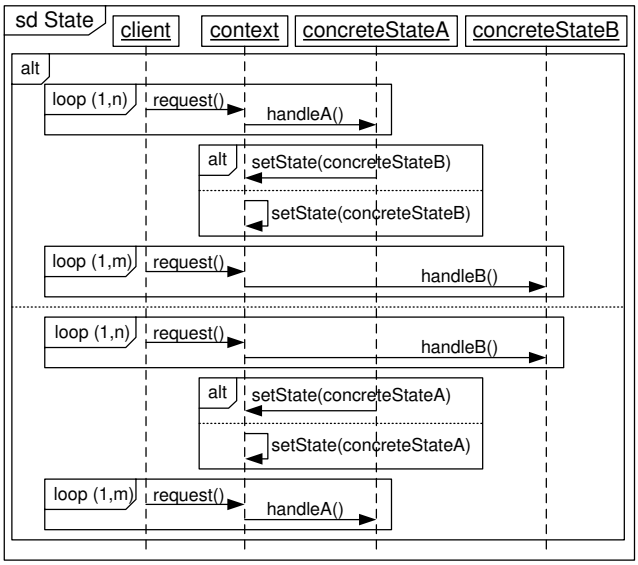

Figure 3: Behavioural pattern for *State*

For the specification of behavioural patterns we use a notation based on UML 2.0 sequence diagrams [8]. As an example, Figure 3 shows the behavioural pattern for the *State* design pattern. The pattern requires the existence of four objects, namely client, context, concreteStateA, and concreteStateB. The pattern describes two alternative sequences. In the first sequence the client object calls the method request on the context object which in turn calls handleA on object concreteStateA. This interaction fragment must occur at least once but may occur an arbitrary number of times which is specified by loop(1,n). Then either the concreteStateA or the context itself has to change the state by calling the setState method with concreteStateB as argument. After the state change the client has to call request on context at least once again. This time the behaviour of context must be handled by the state concreteStateB. This specification conforms to the behavioural description of the *State* pattern [1]. In principle the second alternative specifies the same behaviour as the first one except that the context is in state concreteStateB first and then changes to concreteStateA.

Note that between and within the specified method calls an arbitrary number of other methods which are not mentioned in the pattern may be called. However, the calls specified by the pattern have to occur in exactly the specified sequence. This conforms to the semantics of the UML 2.0 *consider* interaction operator which implicitly holds for all behavioural patterns. To facilitate a more restrictive specification we also support the *critical* operator which may be assigned to interaction fragments to prohibit method calls which are not specified explicitly.

To verify the conformance of a pattern candidate to its corresponding behavioural pattern we record method traces during the execution of the program in which the interaction specified by the behavioural pattern has to be recognized.

A FUJABA plug-in for the specification of the behavioural patterns is currently being developed and a plug-in performing the actual dynamic analysis will follow. A new plug-in for the recording of method traces in Java programs and its integration into the approach is presented in the following section. This plug-in will be used by the dynamic analysis to record the traces.

3. Selective Tracing

Recording all method traces during a program's execution produces too much information. Furthermore, the monitoring of a complete program extremely reduces the runtime performance. For most analyses a "slice" of all method traces is sufficient. In this approach the static analysis provides a set of pattern instance candidates that has to be further analysed by dynamic analysis. All other classes of the program can be ignored.

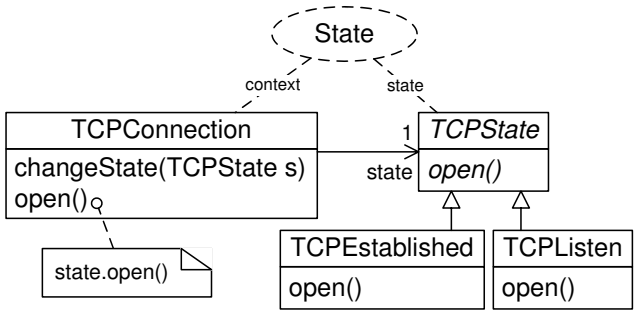

Figure 4: Example of a *State* instance

Figure 4 shows an example for a *State* candidate. It has been recovered and annotated by the static analysis. For the dynamic analysis the method traces for the candidate have to be recorded. This is done by the new FUJABA plug-in JAVATRACER.

Input for JavaTracer

The input for the JAVATRACER is given as an XML document in which the candidate's classes and some of their methods are listed that have to be monitored during program execution. This information is retrieved from the candidate and the structural and behavioural patterns.

The classes to be monitored can be gathered from the behavioural pattern in Figure 3. There are three objects within the sequence diagram on which methods are called, namely context, concreteStateA and concreteStateB. The names of these three objects refer to the nodes context:Class, concreteStateA:Class and concreteStateB:Class within the structural pattern in Figure 2. During static analysis the nodes from the structural pattern have been mapped to the nodes of the candidate in Figure 4. By using this mapping we can extract the classes from the candidate that have to be monitored, namely TCPConnection, TCPEstablished and TCPListen.

The methods can be extracted in the same way. In Figure 3 the four different methods request, setState, handleA and handleB are called. They refer to request:Method, setState:Method, handleA:Method and handleB:Method from the structural pattern. They have been mapped to the methods TCPConnection.open(), TCPConnection.changeState(TCPState s), TCPEstablished.open() and TCPListen.open().

The JAVATRACER can also restrict the recording of method calls to a given caller. The handleA and handleB methods in Figure 3 are called by the context object. So the caller for the TCPEstablished.open() and TCPListen.open() methods is the TCPConnection class. The method setState in the behavioural pattern is called by three different objects. So for the method TCPConnection.changeState() the three caller classes TCPConnection, TCPEstablished and TCPListen have to be monitored.

```
<Trace>
  ...
  <ConsiderTrace>
    <Class name="TCPConnection">
      <Method name="open"/>
      <Method name="changeState">
        <Parameter type="TCPState"/>
        <Caller name="TCPConnection"/>
        <Caller name="TCPEstablished"/>
        <Caller name="TCPListen"/>
      </Method>
    </Class>
    <Class name="TCPEstablished">
      <Method name="open">
        <Caller name="TCPConnection"/>
      </Method>
    </Class>
    <Class name="TCPListen">
      <Method name="open">
        <Caller name="TCPConnection"/>
      </Method>
    </Class>
  </ConsiderTrace>
  ...
</Trace>
```

Figure 5: Example of JavaTracer input

Figure 5 shows an excerpt of the input for the JAVATRACER. The candidate's classes given in the input will be monitored using the *consider* semantics, i.e. only the given methods will be monitored, method calls of other methods will be ignored. These classes are listed within the ConsiderTrace section of the input.

The JavaTracer also provides *critical* monitoring of classes where all methods of a class are monitored. This facilitates the checking of critical method call sequences. The classes are specified within a CriticalTrace section of the input.

Tracing

The JAVATRACER acts as a debugger and executes the program to be analysed, called the debuggee. It uses the Java Debugging Interface (JDI) [5] for connecting to the debuggee's virtual machine. For each method given in the input two breakpoints are set at the beginning and the end of the method body. The JAVATRACER is informed, when a breakpoint is reached during program execution.

This approach is not bound to Java even though the JAVATRACER is implemented for Java programs only. Breakpoints are a common feature of debuggers for nearly all languages. So in principle a selective tracer for different languages can be implemented in the same way.

When the debuggee reaches a breakpoint the JAVATRACER will be informed. The JAVATRACER halts the debuggee and asks the debuggee's virtual machine for additional information about the method call. This includes information about the method name, the time stamp for the method call, the names and unique identifiers of the caller and callee objects, the identifiers of objects passed as arguments as well as the current thread. Then the debuggee's execution

is continued.

The execution of the program is controlled either manually by the reengineer or by automated tests. The JavaTracer informs the reengineer which classes have been loaded and which methods have been executed.

Output of JavaTracer

Figure 6 shows an excerpt from the JavaTracer's output. The output consists of a list of method entry and exit events in the order of their occurrence.

```
<TraceResult>
  ...
  <TraceEvent time="1089792972829">
    <Callee id="3">
      <Object objectName="TCPConnection" uniqueID="42"/>
      <Method methodName="open"/>
    </Callee>
  </TraceEvent>
  <TraceEvent time="1089792972830">
    <Callee id="15">
      <Object objectName="TCPEstablished" uniqueID="48"/>
      <Method methodName="open"/>
    </Callee>
    <Caller>
      <Object objectName="TCPConnection" uniqueID="42"/>
      <Method methodName="open"/>
    </Caller>
  </TraceEvent>
  <TraceEvent time="1089792972845">
    <MethodExit id="15">
      <Method methodName="open"/>
    </MethodExit>
  </TraceEvent>
  ...
</TraceResult>
```

Figure 6: Example of JavaTracer output

The three trace events describe a call of method open on an object of class TCPConnection. This method calls another method open on an object of class TCPEstablished. The last method call immediately returns. These three events cover the first loop within the behavioural pattern of Figure 3.

4. Related Work

The JaVis environment [3] visualizes and debugs concurrent Java programs to detect deadlocks. The information about a running program is collected by tracing, which is implemented using the JDI [5]. However, this approach uses another technique of the JDI. The debugger has to provide a filter, which specifies the classes and methods to be monitored. During the debuggee's execution all classes and all methods are monitored. For methods passing the filter *MethodEntry-* and *MethodExitEvents* are sent to the debugger. Since all methods are monitored this technique can slow down the debuggee up to 10.000 times.

The Omniscient Debugger [2] records method calls and variable state changes of Java programs. It instruments the source code on the byte code level, i.e. additional code is inserted into the original source code of the debuggee. The code is used to inform the debugger about method calls. The instrumentation is also done in a non-selective way. The author reports about 100MB/sec of information produced during the execution as the main problem of this approach.

5. Performance

Table 1 shows the performance of different executions of the Fujaba Tool Suite. In the first case the duration of starting Fujaba was measured[2]. In the second and the third case a project was opened in Fujaba. The first project consists of one class diagram with 12 classes, the second one of one class diagram with 27 classes and 178 activity diagrams. Four major classes were monitored.

Action	$t_{w/o}$	t_{break}	t_{events}
Starting Fujaba	5,39 sec.	8,4 sec.	103,37 sec.
Open Project I	2,85 sec.	20,65 sec.	241,78 sec.
Open Project II	6,28 sec.	49,58 sec.	923,03 sec.

Table 1: Duration of program tracings

First, the program was executed without any tracing ($t_{w/o}$). Then, the program was monitored using the breakpoint approach (t_{break}) and at last by filtering the *MethodEntry-* and *MethodExitEvents* (t_{events}). The table shows that selective tracing with breakpoints improves the performance significantly compared to the event based approach.

6. Future Work

We are currently implementing a Fujaba plug-in for the specification of behavioural patterns. Next, the recognition of the behavioural patterns in the method traces will be implemented using basically the same techniques as in static analysis. The behavioural patterns will be translated into graph grammar rules. The output from the JavaTracer will be transformed into a method call graph for each candidate. If the graph grammar rule for the behavioural pattern can be applied to the call graph, the candidate can be verified as a correct design pattern instance.

References

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

[2] B. Lewis. Recording events to analyze programs. In *Object-Oriented Technology. ECOOP 2003 Workshop Reader*. Lecture notes on computer science (LNCS 3013), Springer, July 2003.

[3] K. Mehner. *JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs*, pages 163–175. LNCS 2269. Springer Verlag, May 2001.

[4] J. Niere, W. Schäfer, J. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.

[5] Sun Microsystems. *Java Platform Debugger Architecture (JPDA). Online at http://java.sun.com/products/jpda/index.jsp.*

[6] University of Paderborn, Germany. *Fujaba Tool Suite. Online at http://www.fujaba.de/.*

[7] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, May 2003.

[8] L. Wendehals. Specifying patterns for dynamic pattern instance recognition with UML 2.0 sequence diagrams. In *Proc. of the* 6^{th} *Workshop Software Reengineering (WSR), Bad Honnef, Germany*, May 2004. to appear.

[2]The analysis was done on 1GHz Athlon, 640MB RAM, Windows 98 2nd edition, JDK 1.4.2

Fujaba based tool development and generic activity mapping: building an eHomeSpecificator

Ulrich Norbisrath, Priit Salumaa, Erhard Schultchen
Department of Computer Science III, RWTH Aachen University,
Ahornstr. 55, 52074 Aachen, Germany
{uno|priit|erhard}@i3.informatik.rwth-aachen.de

ABSTRACT

To achieve a wide application and acceptance of eHome technology, it must be easy to install value-added services to all different kind of households. On the one hand, the development process to adapt different services to a particular household must be minimized and on the other hand the configuration and deployment process for eHome systems must be automated. To achieve such an automation, the differences between various eHomes and desired value-added services have to be specified in a machine-readable form. With the aid of Fujaba, we created an eHome model capable of specifying functions, devices, environments, and value-added services. The static and dynamic aspects of the model are completely defined in Fujaba. To use the model and to do the actual specification for a particular eHome and particular appliances we generated the eHomeSpecificator tool from the model. The dynamics of the model in the form of activities is integrated in the developed tool with the help of a generic mechanism, which will be presented here. To help integrating the eHomeSpecificator in the configuration and deployment chain for assembling eHome systems we developed OWL translators to support our knowledge base.

1. INTRODUCTION

The term *eHome* denotes a home which offers through the combination of its electronic equipment advanced benefits as *value-added services* for its inhabitants. Systems implementing the value-added services are called *eHome systems*. To achieve a wide application and acceptance of eHome technology, it must be easy to install value-added services to all different kind of households. Current approaches to equip eHomes with value-added services require a new development step for adapting services to meet the constraints of the particular eHome. To automate configuration and deployment of value-added services, we consider the following tool chain: An eHomeSpecificator is used to specify functions, devices, environments, and value-added services. This information is used as input for a Deployment Producer [4], which supports the generation of a description from this particular specification and information from a knowledge base. usable for the deployment of the desired value-added services. The deployment is carried out by the Runtime Instancer, which uses the particular specification generated by the Deployment Producer. The eHomeSpecificator is realized with the help of Fujaba [5] and will be presented here. We choosed Fujaba to address frequent changes in our model to speed-up the development process. According to that the focus is to show that graph rewriting language based software development is a successful way to develop necessary tools to support eHome systems.

As a concrete example, we consider a customer living in a partly equipped eHome-apartment (see figure 1). In the apartment, there are six rooms: corridor, office, living room, bathroom (including toilet), kitchen, and bedroom. Each of these rooms has at least one controllable light source and is connected via one door to the corridor. The corridor itself has an entrance door. One door leads from the kitchen to the living room. Every room except the corridor has a window. The windows in the bedroom and the living room are equipped with window breakage sensors. At the entrance and in the corridor, open/ close-sensors are installed, and there is one camera in the corridor. The kitchen is equipped with a fire sensor. The customer living in this apartment desires an automated lighting system and surveillance of his/ her property for added security. These requests may be fulfilled with the help of two value-added services: light movement and security. The eHomeSpecificator helps to specify, how the location structure of an eHome environment looks like, which devices are already installed, and what the requirements of a particular value-added service are. It also helps to add new devices necessary for fullfilment of desired value-added services.

This work summarizes various aspects from [8] but outlines the technical work related to Fujaba and UPGRADE more thoroughly.

2. FROM FUJABA TO EHOMESPECIFICATOR

In our project, we use Fujaba to specify the data-model in terms of UML class-diagrams. Activities are added to implement logical components of the model. We choose not to integrate our project into the Fujaba framework itself, but to create a stand-alone application that is based only upon the Java code generated by Fujaba. We will refer to this application containing the Fujaba data-model code as the eHomeSpecificator.

This means some repetitive effort regarding basic user interaction compared to a plug-in development project. However, this stand-alone approach allows greater flexibility as it does not pose any restrictions regarding user-interaction. Also, developers creating the data-model and system logic are strictly separated from the tool's users: Developers use Fujaba to maintain the specification and generate Java code that is easily integrated into the tool.

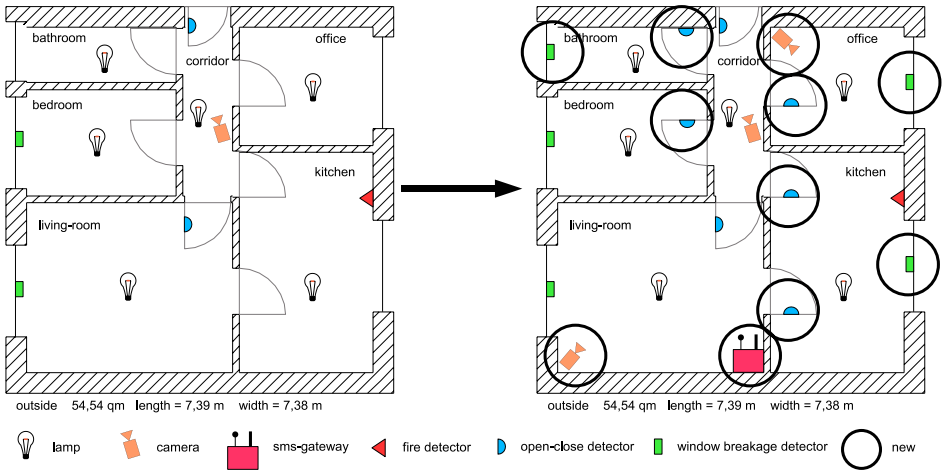

Figure 1: Example for applying the security scenario.

We are concentrating on easing up the code transition from Fujaba to the eHomeSpecificator. In particular, we focus on making activities defined in the UML model available in the eHomeSpecificator user-interface in a generic yet helpful way. For example, the generated and compiled Java code created from an activity diagram does not contain useful names for its parameters or for the activity itself. We add an XML description file that defines additional properties for each activity, such as labels and tooltips (for example see listing 1).

```
<ACTIVITY name="createDeviceFunction"
          label="New DevFunc">
  <TOOLTIP>Adds a device function to the
  selected device definition.</TOOLTIP>
  <CONTEXTS><DEVICEDEFINITION /></CONTEXTS>
  <PARAM label="Function Class">
    <TOOLTIP>The function class of the new
    device function.</TOOLTIP>
  </PARAM>
  <PARAM label="Input"></PARAM>
  <PARAM label="Output"></PARAM>
</ACTIVITY>
```

Listing 1: Example of an XML Activity Specification for Generic Representation Mechanism

Here, we present only a fractional part of the model for eHomes. It covers the specification of device definitions, which are used later in the specification of the given eHome. On this restricted example we illustrate how the model relates to Fujaba and the eHomeSpecificator. The development of the eHomeSpecificator begins with the design of the eHome model in Fujaba. The static structure of the model is specified in the UML class diagram (see figure 2) and its dynamic aspect is captured by activities specified with Fujaba activity diagrams (see figure 3).

As mentioned, figure 2 presents only necessary information for defining devices. A device could be any kind of appliance like a fire detector, a lamp, or a software component in the form of a controller. The device definitions combine functions like heat or smoke detection, which are defined by function classes and are available over the device interfaces. According to the interfaces the devices can later be connected in the eHome specification or eHome system deployment. The graphical representation of the definition of the fire detector device in the eHomeSpecificator tool is presented in figure 4.

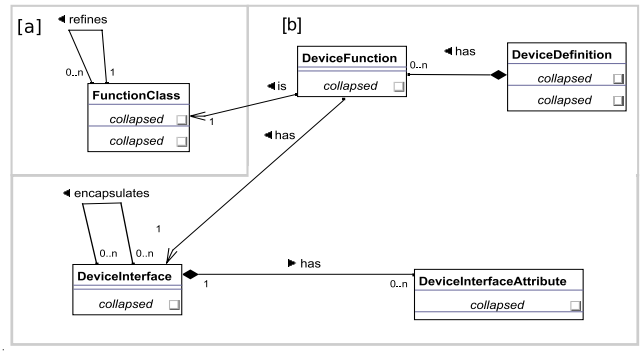

Figure 2: Portion of eHome Model

Designed activities of the eHome model are the only means how to modify the model and to get information about it. In figure 3 the activity for adding a function to a device definition is presented. The activity creates a new `DeviceFunction` for the given device definition referred to as the `this` object on the diagram. The created `DeviceFunction` is from the given `FunctionClass`. The activity is integrated into the tool using the generic activity invocation mechanism mentioned earlier. As a result of the integration the activity appears as a button on the left side of the editor window. This particular activity appears as a button `New DevFunc` in the eHomeSpecificator (see figure 4).

After designing the eHome model in Fujaba, the model is materialized in Java code using Fujaba's code generator. As the next development stage the eHomeSpecificator tool is developed. The tool can be considered as a container and runtime environment for the described eHome model. The activities on the model are bound with the graphical user interface of the tool using the generic mechanism mentioned previously. Different logical parts of the model structure

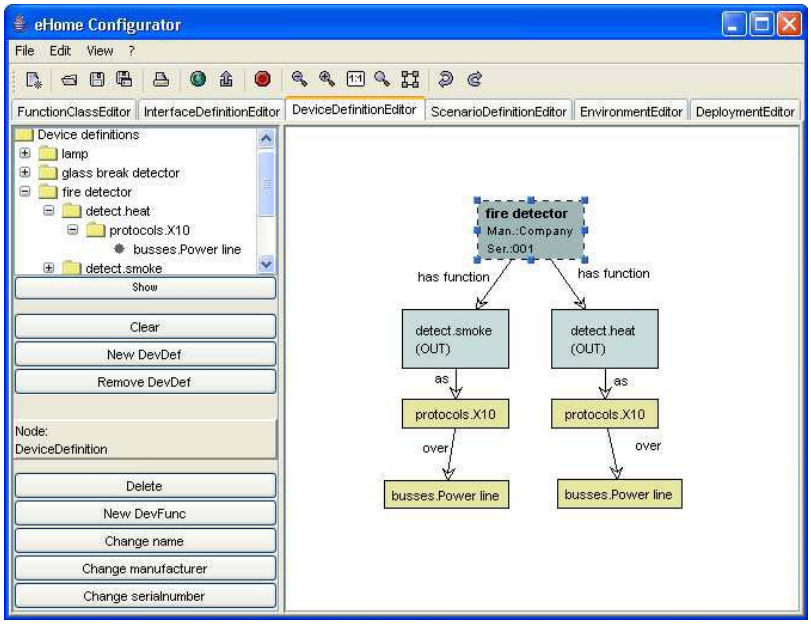

Figure 4: The fire detector in the DeviceDefinition editor.

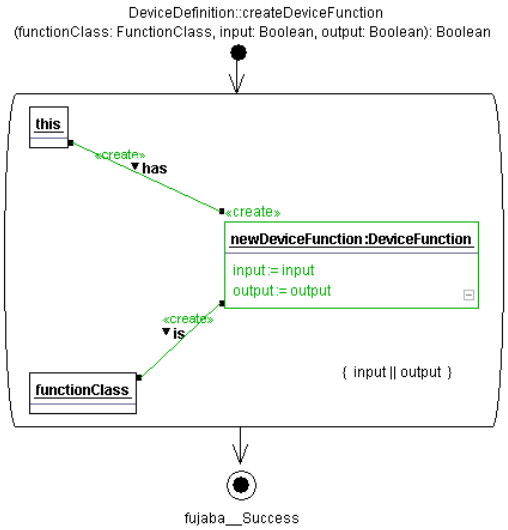

Figure 3: An activity for adding a functionality to the DeviceDefinition.

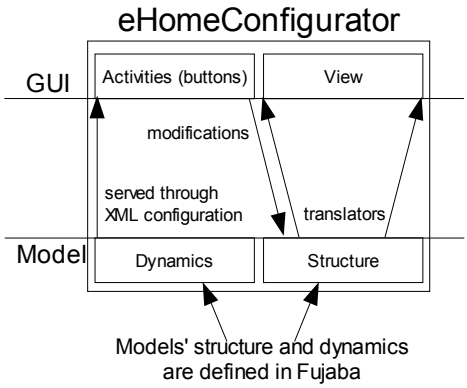

Figure 5: eHomeSpecificator Architecture.

are visualized in the corresponding editors (see figure 4). The logical parts are presented in the user interface using the JGraph [2] technology. There is a translator implemented for each editor view. The task of a translator is to traverse the eHome models' runtime subgraph, relevant to the specific editor and generate the corresponding JGraph to display. An end-user of the tool can derive the specific model for his/ her eHome using different editors and activities presented in editors. In figure 5 we illustrate the relation between the eHome model, the Fujaba specification tool, and the eHomeSpecificator previously presented in this section.

The use of Fujaba gives us several advantages. We can model the inner data structure of the tool, the static structure of the eHome model in the standardized visual specification language UML. Although the acquired class diagram represents a static aspect of the eHome model, we can also visually specify the dynamic side of the model with Fujaba activity diagrams. The generated code from the model is executable and can be easily integrated into the tool using the generic activity invocation mechanism. In the case of a new activity in the model the resulting integration effort consists of code generation, editing the XML file, and compilation of the generated code. In this case, integration is a matter of five minutes. In the case of structural changes, we still have to program by hand the changes for the translators. This overhead will be tackled in the future work.

Since the eHomeSpecificator covers the specification and only to some degree the configuration part of the eHome configuration and

deployment process, it needs to be integrated with the Deployment Producer, which creates with the help of the knowledge base the complete configuration to be deployed into the eHome. For integration with the Deployment Producer, the eHomeSpecificator has to be able to output an OWL eHome ontlogy, which corresponds to the current runtime version of the eHome model. For these means, like also for visualisation of the model, a set of translators is built. These translators traverse the eHome model and generate OWL structures with the help of the Jena Semantic Web Framework[6].

3. RELATED WORK

There are many different tools for visual modeling and code generation. Fujaba is one of them. Another system we could have used is the PROGRES (PROgrammed Graph REwriting) language [9] that was developed at our department. The PROGRES system is not necessarily restricted to UML models, although UML modeling is possible. PROGRES is used to specify and test graph models and rewriting rules, and finally to generate source code that can be executed in an efficient way. It is supported by the UPGRADE framework (Universal Platform for GRAph-based DEvelopment) [1] that can be used to rapidly develop prototypes which operate on the generated code. UPGRADE provides a user-interface that may be adapted to the application's needs. Also, UPGRADE manages a persistent graph storage without the need for any support by the developer. Various projects at our department profited by the combination of PROGRES and UPGRADE to implement a specification-based prototype, for example AHEAD [3]. Fujaba lacks this kind of rapid-prototyping and therefore we had to implement the eHome-Configurator for this purpose. However, Fujaba is widely accepted among developers and in education. It is supported by an active community and can be used free of charge.

4. FUTURE WORK

The eHomeConfigurator was built for the designated purpose of modeling in the eHome context. However, its flexibility allows the eHomeConfigurator do be adapted to other fields of work.

Beside exchanging the data-model created by Fujaba, the developer has to write a new XML description file for the model's activity diagrams. Also, it is required to implement a so-called "Translator" class in Java that aids in displaying the model graph. Currently, displaying the graph including node- and edge-labeling is not and cannot be done automatically, due to lack of information. For example, a node cannot be assigned a suitable label without further information. One could assign the class name of the graph object itself which would be not very helpful to the user. Alternatively, one could enforce the developer to provide a `getLabel` method that delivers an appropriate label text. However, this would require to implement code into the data model that is strictly used to display informations to the user which is not what logical parts of the model are meant for. Also, labels would not be customizable by the user.

With the help of some additional informations that are, as the activity description already is, placed in an XML description file, labeling the graph can be automated. For example, this description could enumerate possible fields of an object that should be used to construct a label. Hence, we could achieve an UPGRADE-like attitude.

For more information on the the eHomeSpecificator and its integration in the configuration and deployment chain see [7].

5. SUMMARY

We apply the graph rewriting language based software development practices on developing a supporting tool for eHome systems. We consider this approach a more productive one as traditional development methods. The developed eHomeSpecificator tool is used to view and edit eHome specifications. It can be embedded with its OWL translators in our configuration and deploament chain. The inner data model and activities of the model were designed and generated with the help of Fujaba. The model was integrated into the tool mostly with the focus on generic mechanisms. The consequence of this is that changes to the model will only require small changes to the tool.

6. REFERENCES

[1] B. Böhlen, D. Jäger, A. Schleicher, and B. Westfechtel. UPGRADE: Building Interactive Tools for Visual Languages. In N. Callaos, L. Hernandez-Encinas, and F. Yetim, editors, *Proceedings of the* 6^{th} *World Multiconference on Systemics, Cybernetics, and Informatics (SCI02)*, volume I (Information Systems Development I), pages 17–22, Orlando, Florida, USA, July 2002. IIIS.

[2] J. Community. Jgraph swing component. `http://jgraph.com/jgraph.html` (01.06.2004).

[3] D. Jäger, A. Schleicher, and B. Westfechtel. AHEAD: A graph-based system for modeling and managing development processes. In M. Nagl, A. Schürr, and M. Münch, editors, *Proceedings Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE'99)*, volume 1779 of *LNCS*, pages 325–339, Kerkrade, The Netherlands, Sept. 2000. Springer.

[4] M. Kirchhof, U. Norbisrath, and C. Skrzypczyk. Towards Automatic Deployment in eHome Systems: Description Language and Tool Support. In *Proceedings of 12th International Conference on Cooperative Information Systems (CoopIS 2004)*, Lecture Notes in Computer Science. Springer, 2004. to appear.

[5] T. Klein, U. Nickel, J. Niere, and A. Zündorf. From uml to java and back again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, September 1999.

[6] H. P. Labs. Jena Download. `http://jena.sourceforge.net/downloads.html` (20.05.2004), 2004.

[7] U. Norbisrath and P. Salumaa. eHomeConfigurator. `http://sourceforge.net/projects/ehomeconfig`, 2004.

[8] U. Norbisrath, P. Salumaa, E. Schultchen, and B. Kraft. Fujaba based tool development for eHome systems. In *Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004. to appear.

[9] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. PhD thesis, RWTH Aachen, 1991.

Simulation and Testing of Mobile Computing Systems using Fujaba

Ping Guo
International Graduate School of Dynamic Intelligent Systems
University of Paderborn, Germany
ping@upb.de

Reiko Heckel*
Department of Computer Science
University of Dortmund, Germany
(on leave from University of Paderborn)
reiko@upb.de

ABSTRACT

The paper presents an approach for analysis, modeling and validation of mobile information systems with the tool support of Fujaba. The approach is developed based on UML-like meta models and graph transformation techniques to support sound methodological principals, formal analysis and refinement. With conceptual and concrete level of modeling and simulation, the approach could support application development and the development of new mobile platforms. The approach also provides automatic analysis, validation and behavior consistency check with the support of Fujaba.

1. INTRODUCTION

Mobility is a "total meltdown" of the stability assumed by distributed systems as stated in [7]. The main difference is caused by the possibility of roaming and wireless connection. Roaming implies that, since devices can move to different locations, their computational context (network access, services, permissions, etc.) may change, and the mobile hosts are resource limited. Wireless connections are generally less reliable, more expensive, and provide smaller bandwidth, and they come in a variety of different technologies and protocols. All these result in a very dynamic software architecture, where configurations and interactions have to be adapted to the changing context and relative location of applications.

Mobility has created additional complexity for computation and coordination, which makes the current architectural concepts and techniques hard to use [2]. The current architectural approach offers only a logical view of change; it does not take the properties of the "physical" distribution topology of locations and communications into account . It relies on the assumption that the computation performed by individual components is irrelative to location of the component, and the coordination mechanisms through connectors can be always transmitted successfully by the underlying communication network. In order to support mobility, the architectural approach needs to be adjusted in different abstract layers of modeling and specification languages.

As shown in [5], there are a lot of platforms and middleware have been developed for mobile computing. These different platforms and middleware provide different transparency levels of context awareness to the application, where the application has to be aware of, and be able to react to, changes in its context given by its current location, quality, cost and types of available connections, etc. The amount of context information required and available to the application greatly varies, depending on the employed infrastructure so that, in the end, not every intended application scenario may have a meaningful realization on any given platform. That means, developers have to take into account the properties of the infrastructure they are using, not only for the final implementation, but also already at a conceptual level during requirement analysis.

A conceptual model capturing the properties of a certain class of mobile computing platforms would be very helpful to the application development and the development of new mobile platforms. It would allow an understanding of the basic mechanisms and their suitability for a certain task. With suitable refinement and evolution support, the conceptual model can be mapped into a concrete platform specific model.

In reality, it is difficult and expensive to test the mobility support of a certain platform, which requires devices supporting wireless communication and specific tools to check the coordination logic of involved hardware and software components. Simulating the mobile platform can provide a simple and cheaper way to test the mobility aspects of the platform. Through this means, the context aspects of the platform like locations, network connections can be simulated directly, thus a dynamic execution environment can be provided for the context-aware applications, which is also difficult to test in reality.

*Research partially supported by the European Research Training Network *SegraVis* (on *Syntactic and Semantic Integration of Visual Modeling Techniques*)

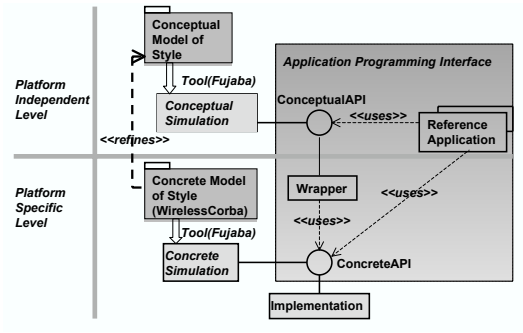

Figure 1: Modeling and simulation framework

In this paper, we introduce our approach for analysis, design and simulation of mobile systems with the tool support of Fujaba. The approach is developed based on UML-like meta models and graph transformation techniques to support sound methodological principals, formal analysis and refinement. The approach includes two main parts: modeling and simulation. The modeling part is introduced in Sect. 2. Simulation is introduced in Sect. 3. Sect. 4. give the related work and Sect. 5 concludes the paper with finished work and future work.

2. MODELING OF THE MOBILE SYSTEM

Conceptual and concrete level modeling (as shown in Fig. 1) are the key parts of our approach [4]. The conceptual modeling of styles of mobile systems [5] is proposed as a way of capturing the properties of a certain class of mobile computing platforms. The conceptual model consists of two parts: a static structural model given by UML class diagrams whose instances represent the valid system configurations, and a dynamic behavioral model given by transformation rules over these instances, specifying the operations of the style. Typed graph transformation systems [8] will provide the underlying formal model and operational semantics. Informally, a typed graph transformation system $G = \langle TG, C, R \rangle$, where TG is a *type graph* (visualized by the class diagram) defining the architectural elements, C is a set of constraints restricting their possible compositions, and R is a set of *graph transformation rules* (given by pairs of object diagrams).

Our structural model consists of meta models at different levels contained in different packages. This allows us to separate different concerns, like software architecture, distribution and roaming, while at the same time retaining an integrated representation where all elements of a concrete model are presented as vertices of the same graph, i.e., an instance of the overall meta model. Based on this uniform representation, the different sub-models can be related by associations between elements belonging to different sub-models.

Based on the integrated representation of the different views in a single meta model, we can define the rules governing movement and connectivity as graph transformation rules typed over the corresponding package(s). A graph transformation rule $r : L \Rightarrow R$ consists of a pair of TG-typed instance graphs L, R such that the intersection $L \cap R$ is well-defined. The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions. In Fig. 2, the *moveIn* rule is shown as an example: according to its precondition, expressed by the pattern on the left-hand side, there should be a Node n and an Area a whose types NT and AT should be connected by a *locatedAt* link. That means the node is of a type that is supported by the area, like a cell phone in a GSM cell. In this case, the rule can be applied with the result of creating a new *locatedAt* link between the two instances. This is expressed in the post-condition of the rule shown on the right-hand side.

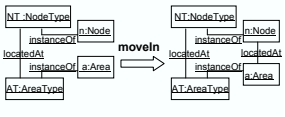

Figure 2: Transformation rule *moveIn*

In [5], we have presented a basic style of mobile information system for nomadic network, which is focussed on the roaming and connectivity of mobile hosts, i.e., hosts can change location and possible connections may vary according to this location change. Naturally, architecture and behavior of applications depend on the connectivity and location of their host computers. Our three-layered meta model captures these relations in the three packages *Architecture*, *Connectivity* and *Roaming* to present different viewpoints of the systems. The basic operations of the style include *moveIn, moveOut, register, deRegister, connect, disconnect* and *handOver*.

The concrete model is based on a specific platform, e.g. Wireless CORBA. Concrete modeling of mobile system uses the same modeling technologies as conceptual modeling of styles. Given specified models of the platform, a prototype can be generated directly for the reference of implementation using code generation functionality provided by graph transformation tools like Fujaba [1].

The relationship between these two different layer models is refinement, e.g., the mobility and other aspects modeled in the conceptual model need to be mapped into a concrete design. Besides this, we focus on the behavior consistency check of the two levels (will be discussed in Sect 3.3.), i.e. the consistency check of the rules applied to the models, but not of the structural elements. This is because the construction elements inside the conceptual model do not need to be present in the concrete model, that makes the consistency check of the construction element not much meaningful in our approach.

3. SIMULATION

The operational semantics of the typed graph transformation system allows us to execute the models thus analyzing the system through simulation. In this section, we will introduce two ways to use the simulation through Fujaba: for validating the model and as an oracle for test the actual implementation.

3.1 Simulation for Validation

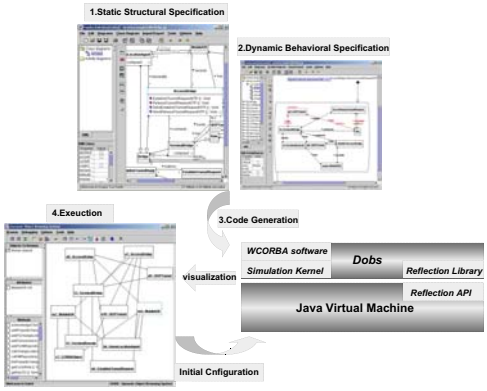

Figure 3: Fujaba Simulation

In graph transformation systems, many verification problems can be formulated as reachability (or non-reachability) properties of a given configuration in the system. A reachability property holds for a given graph transformation system $G = \langle TG, C, R \rangle$ and a start graph G_0 if an instance graph that contains a certain target pattern is reachable by applying available transformation rules. This means that a system can evolve from the start configuration to the desired target configuration by performing given operations. In this way we can check, for example, if a required reference application scenario is realizable on the middleware, thus validating functional completeness of the model.

The object-oriented CASE tool Fujaba [1] supports the specification of a system using UML class diagrams and story diagrams, a combination of activity diagrams and collaboration activity diagrams(as a notation for graph rewriting rules). Executable Java source code can be generated automatically. To observe the running system, a Dynamic Object Browsing system (Dobs) supports the execution of rules, visualizing the effect of the application on an object graph representing the state of the Java heap.

We introduce how to use Fujaba to validate our defined specification. As shown in Fig. 3, we use the Fujaba class diagram editor to specify our meta model at first. Graph transformation rules (or operations) are defined in the Fujaba story diagram editor then. After generating and compiling Java code for the complete specification, we can start Dobs to execute the models. We can create an initial object configuration typed over the defined class diagrams. The initial object configuration represents a possible configuration of the system, which is also a start graph G_0 as defined before. Following the defined sequence of operations that describe the application scenario, we can then execute the sequence of operations starting from G_0. For example, we can test if the terminal-initiated handoff scenario [3](defined by a sequence of operations)is reachable by starting from the initial configuration. Through this way, we can test if the pre-defined scenario is supported by our specification, thus validate the functionality completeness of our model.

3.2 Simulation for Testing

All software testing methods depend on the availability of an oracle, that is, some methods for checking whether the system under test has behaved correctly on a particular execution. Executable formal specifications can be used as test oracles to produce the results expected for a test case. By comparing the result of a call to the actual implementation with the result of a call to the simulation, the test oracle can be used to check the correct execution of an operation.

We can extend the specified concrete model to a test oracle [3]. Since the concrete model is platform independent concerning the independency of specific programming languages, hardware platforms and concrete implementation methods, it can be reused as a reference to test the correctness of implementations on different platforms. As a test driver, a standard reference application shall be required. To facilitate the interaction between the reference application with our model (resp., the code generated from it), we need to provide an Application Programming Interface(API) that is consistent to the API provided by a middleware implementation. Using the same test application as a test driver for the implementation and for the defined model, the developers can trace errors in the execution and check the pre- and post-conditions of operations.

3.3 Wrapper for API–Behavior Consistency

The correct refinement of abstract conceptual styles into a concrete style is important, and the verification process is usually complicated. In order to automatize the consistency check between the conceptual and concrete models, we develop a Wrapper (in Fig. 1) to define the refinement relationship between these two models. Both the conceptual and the concrete model provide application programming interfaces through the operations defined via the rules, which are named Concrete API and Conceptual API as shown in Figure 1. As an adapter between Concrete API and Conceptual API, the wrapper encapsulates and maps the operations implemented in Concrete API to the operations defined in Conceptual API. Providing type transformation and semantic match, the Wrapper forwards operation calls to Conceptual API to the operation calls to Concrete API. In this way, the application can use the more abstract interface while the concrete operations offered by the platform remain hidden. This abstraction allows us to port the application to a new concrete platform API by means of a new wrapper, without changing the application itself. The wrapper can be also used to test, e.g., by means of a reference application, if the operations provided by Concrete API and Conceptual API are semantically compatible, therefore verifying the concrete style or the actual platform against the requirements expressed in the conceptual style.

4. RELATED WORK

Several proposals have influenced our approach. The general idea of modeling classes of systems with common structural and behavioral characteristics by a combination of metamodeling and graph transformation is due to [6], where it has been applied to software architecture styles. As mentioned before, the architectural style offers only a "logical" view of change; it does not take into account the properties of the "physical" distribution topology of locations and communication links. In our model, we extend the architecture style by adding mobility aspects, with the focus on roaming and connectivity issues.

Some of the techniques proposed by the AGILE project presented in [2] are close to our approach of modeling. AGILE develops an architectural approach by extending existing specification languages and methods to support mobility: UML stereotypes are used to extend UML class, sequence and activity diagrams in order to describe how mobile objects can migrate from one host to another, and how they can be hosts to other mobile objects. Graph transformation systems are proposed as a means to give an operational semantics to these extensions.

Other extensions are based on architectural description languages, like the parallel program design language CommUnity using graph transformation to describe the dynamic reconfiguration; Klaim as a programming language with coordination mechanisms for mobile components, services and resources; The specification language CASL as a means for providing architectural specification and verification mechanisms.

While Klaim and CASL are more programming and verification oriented, the approaches based on UML and CommUnity are at a level of abstraction similar to ours, but the goals are different: Our focus is to model a style of mobile applications, e.g., corresponding to a certain mobility platform, while the focus in the cited approaches is on the modeling of applications within a style more or less determined by the formalisms. Indeed, being based on a meta-model, our approach can easily specify styles exhibiting all kinds of features like QoS (as demonstrated in [5]) or more sophisticated aspects of context awareness, handOver operations within one or between different networks, etc.

Finally, our three-layered modeling approach provides a clear separation of the different views of software architecture, connectivity, and mobility, which is required in order to specify a physical phenomenon, like the loss of a signal, in relation with the intended reaction of an application or middleware platform, like the transfer of ongoing sessions to a new connection.

The idea of analysis and design of a system using a refinement approach is not new. Generally, the people focus on architecture refinement based on component and connectors where the construction elements in different abstract layers have direct corresponding relationship. In our framework, the construction elements inside the conceptual level do not need to appear in concrete level. We focus on the behavioral consistency check between conceptual and concrete level; this makes it easier to implement automatic consistency checks via testing.

5. CONCLUSION AND FUTURE WORK

The paper presents an approach for analysis, modeling and validation of mobile information systems with the tool support of Fujaba. Presenting the basic structures and operations common to a certain class of mobile computing platforms, with the refinement between conceptual and concrete models, the approach could support application development and the development of new mobile platforms. The approach also provides automatic analyze, validation and behavior consistency check with the support of Fujaba.

We have finished a conceptual model of a style for mobile system, which is a very important part of our approach. Not tailored towards a particular platform, the model reflects the properties of nomadic network, where mobile devices are supported by a fixed infrastructure. The validity of the model has been checked through the Fujaba simulation environment using an application scenario, where the explicitly modeled mobility and context aspects like locations, network connections can be simulated. Some parts of this work are presented in [5]. A concrete model of a specific platform (Wireless CORBA) has been also developed; the models are validated using Fujaba too [3].

A major issue of our future work is the improvement of the simulation to provide a more automated environment for the activation of methods. For example, it would be helpful to derive a sequence of operations from a sequence diagram execute it automatically based on a given the initial configuration. An improved, domain-specific visualization of object configurations is another aim, since the generic object-oriented representation is not concise enough for larger examples. The Wrapper for refinement behavioral consistency check between the conceptual and concrete layers need to be developed, which can be made as Fujaba plugin [1].

6. REFERENCES

[1] From UML to Java and Back Again: The Fujaba homepage. `www.upb.de/cs/isileit`.

[2] L. Andrade, P. Baldan, and H. Baumeister. AGILE: Software architecture for mobility. In *Recent Trends in Algebraic Develeopment, 16th Intl. Workshop (WADT 2002)*, volume 2755 of *LNCS*, Frauenchiemsee, 2003. Springer-Verlag.

[3] P. Guo and R. Heckel. Model-based simulation and testing of mobile middleware: A case study based on wireless corba. submitted.

[4] P. Guo and R. Heckel. Modeling and simulation of context-aware mobile systems. In *Doctoral Symposium of 19th IEEE International Conference on Automated Software Engineering (ASE)*, 2004. Linz, Austria, to appear.

[5] R. Heckel and P. Guo. Conceptual modeling of styles for mobile systems: A layered approach based on graph transformation. In *Working IFIP Conference on Mobile Information Systems(MOBIS)*, 2004. Oslo, Norway, to appear.

[6] Le Métayer, D. Software architecture styles as graph grammars. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 216 of *ACM Software Engineering Notes*, pages 15–23, New York, Oct. 16–18 1996. ACM Press.

[7] G.-C. Roman, G. P. Picco, and A. L. Murphy. Software engineering for mobility: A roadmap. In A. Finkelstein, editor, *Proc. ICSE 2000: The Future of Software Engineering*, pages 241– 258. ACM Press, 2000.

[8] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.

Design and Simulation of Self-Optimizing Mechatronic Systems with Fujaba and CAMeL*

Sven Burmester†, Holger Giese, and Florian Klein†
Software Engineering Group, Warburger Str. 100, D-33098 Paderborn, Germany
[burmi|hg|fklein] @upb.de

ABSTRACT

Self-Optimizing mechatronic systems which are able to react autonomously and flexibly to changing environments are one promising approach for the next generation of mechanical engineering systems. To render designing such systems possible, an approach is required which goes far beyond what is offered by today's standard tools for mechatronic systems. In this paper, we outline how a smooth integration between mechanical and software engineering methods and tools supports the design of verifiable, complex, reconfigurable mechatronic systems. The focus of the paper is on enabling the design and simulation of safe reconfigurable mechatronic systems, as reconfiguration is a critical prerequisite for self-optimization.

1. INTRODUCTION

Mechatronic systems combine technologies from mechanical and electrical engineering as well as from computer science. In the future they will rather be composed of interacting systems than isolated solutions for individual devices. Networking and ever increasing local computational power enable sophisticated mechatronic systems, which, besides more advanced digital control, will include rather complex software coordination and information management capabilities. To handle the resulting complexity, each single unit of such composite systems must be able to react autonomously and flexibly to changing environmental settings.

To achieve the required flexibility, we propose to build self-optimizing technical systems which modify their goals endogenously based on changing environmental settings.[1] A critical prerequisite to realize a goal-compliant autonomous adaptation of the system behavior is the ability of the system to reconfigure its structure or parameters accordingly. This requires coordination between the mechanical engineering and software engineering elements of the system. Therefore a smooth integration between methods and tools from both domains is inevitable. The presented solution integrates the CASE tool Fujaba Real-Time Tool Suite[2] and the CAE tool CAMeL[3] to support the design of verifiable, complex, reconfigurable mechatronic systems.

The paper is organized as follows: In Section 2, the semantic integration between block diagrams for mechatronic control systems and UML component models is summarized. Section 3 presents the necessary extensions to CAMeL (3.1) and Fujaba (3.2) and the binding tool and runtime environment required for ultimately integrating and executing the model (3.3).We then review relevant related work in Section 4 and present our final conclusions and give a short outlook on planed future work.

2. SEMANTIC INTEGRATION

As a running example, we will use the active vehicle suspension system for the shuttles from the *RailCab*[4] research project. In this project, a modular rail system will be developed; it is to combine modern chassis technology with the advantages of the linear drive technology (as applied in the Transrapid[5]) and the use of existing rail tracks. In our example, shuttle software is developed that realizes the safe switching between three different feedback controller structures, which control the body of the shuttle.

2.1 Control Engineering

Feedback controllers are usually specified through block diagrams or differential equations [11] and describe the relation between continuous in- and output signals. In our example, three different feedback controllers are applied, providing different levels of comfort to the passengers:

The controller reference uses the absolute acceleration of the coach body $\ddot{z}_{abs}$ and a reference trajectory that describes the motion of the coach body z_{ref} as input signals. In case the trajectory is not available, the absolute controller, requiring only $\ddot{z}_{abs}$, has to be used. If neither the trajectory nor the measurement of $\ddot{z}_{abs}$ are available, the robust controller is applied, requiring just common inputs (see Figure 1).

For switching between two controllers, one must distinguish two cases: When switching between the normal and the failure block in Figure 1, the change can take place between two computation steps (*atomic switching*). Switching between reference and absolute requires *cross-fading* in order to guarantee stability. The cross fading itself is specified by a fading function $f_{switch}(t)$ and an additional parameter which determines the duration of the cross fading.

†Supported by the International Graduate School of Dynamic Intelligent Systems. University of Paderborn
*This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.
[1]www.sfb614.de/eng/
[2]www.fujaba.de
[3]www.ixtronics.de
[4]http://www-nbp.upb.de/en
[5]http://www.transrapid.de/en

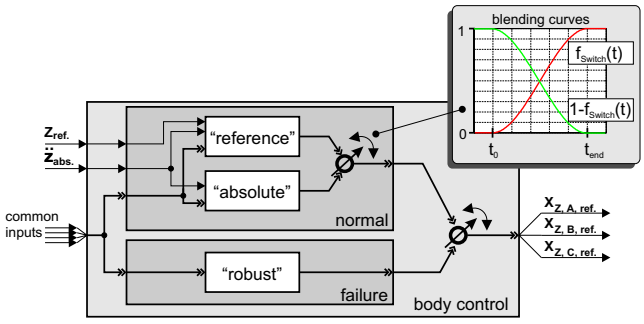

Figure 1: Different control modes and fading

2.2 Software Engineering

Inspired by ROOM [13], UML 2.0 supports the specification of the structure of complex systems using components with ports and deployment diagrams. The only support for real-time behavior is provided by the Profile for Schedulability, Performance, and Time [12]. In order to specify real-time behavior, we apply the real-time extension, the so called *Real-Time Statecharts* in Fujaba [2] as well as a restricted notion for Real-Time Patterns [5].

2.3 Integration

As proposed by the UML 2.0 approach, we use component diagrams to specify the overall structure of the system. We introduce *hybrid components* [4, 3], which integrate discrete and continuous behavior. To model communication of sporadic events and communication of continuously changing signals, we distinguish between discrete and continuous ports. The latter are visualized by a triangle inside the port-square, indicating the direction of the data flow.

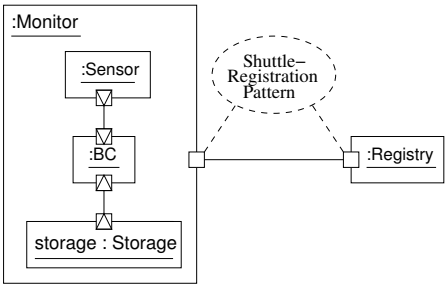

Figure 2: Monitor and its environment

In Figure 2, the structure of the shuttle's Monitor component is shown. It consists of the Sensor, delivering $\ddot{z}_{abs}$, the Body Control (BC) component, switching between the feedback controllers, and the Storage component used for storing the reference trajectory the Monitor obtains from a track section's Registry. A more detailed description can be found in [4, 3].

The Shuttle-Registration communication pattern in Figure 2 specifies the communication protocol between two components. Real-time model checking is used to verify the protocol. Compositional model checking and refinement relations enable even the verification of large, complex systems [5].

The behavior of hybrid components is specified using our notion of Hybrid Statecharts [4, 3], which extend the Real-Time Statecharts. In Hybrid Statecharts, each discrete state

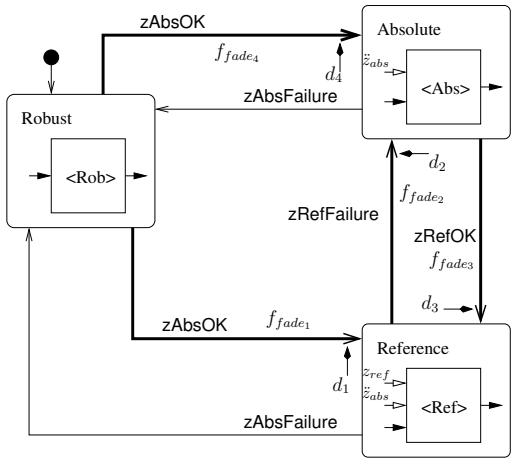

Figure 3: Behavior of the body control component

is associated with a configuration of embedded components. Figure 3 shows the behavior of the BC component as a simple example where each configuration consists of just one continuous feedback controller from Section 2.1.

State changes in Hybrid Statecharts are either modeled through *atomic* or *fading transitions*. The latter (visualized by thick arrows) can be associated with a fading function (f_{fade}) and the required fading duration interval $d = [d_{low}, d_{up}]$ specifying the minimum and maximum duration of the fading.

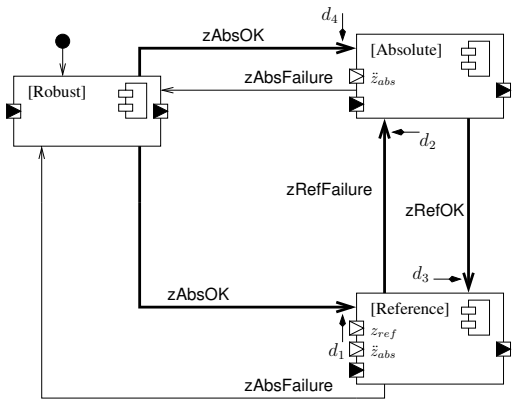

Figure 4: Interface Statechart of the BC component

As reconfiguration leads to changing interfaces (e.g. BC's continuous input signals are state-dependent), we provide the notion of hybrid *Interface Statecharts* (see Figure 4). They only consist of the externally relevant real-time information (discrete states, their continuous in- and outputs, possible state changes, their durations, signals to initiate transitions, and signal flow information [9]). They abstract from the embedded components and from the fading functions. Ports that are required in each of the three interfaces are filled in black.

Well-known approaches like hybrid automata [1] or HyCharts [6] embed only one continuous component, just as the Hybrid Statechart from Figure 3. This is insufficient, however, if reconfiguration is supposed to be possible at multiple levels, which requires hybrid components and their re-

configuration rather than merely the reconfiguration of the controllers.

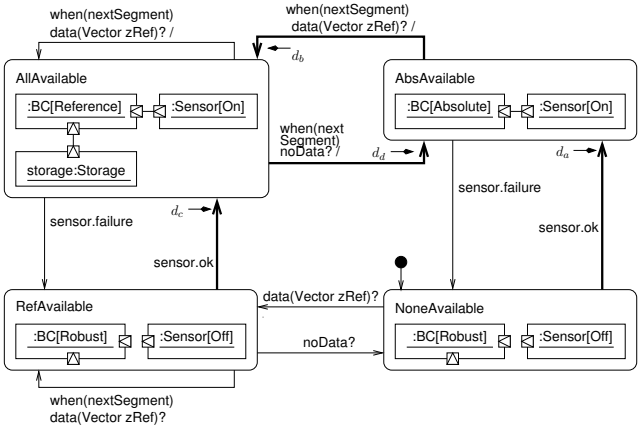

Figure 5: Monitor behavior with modular reconfiguration of the subcomponent BC

In our example, the Hybrid Statechart from Figure 5 specifies the behavior and the reconfiguration of the monitor component. It consists of four discrete states indicating which of the two signals $\ddot{z}_{abs}$ and z_{ref} are available. Every discrete state has been associated with a configuration of the subcomponents BC, Sensor, and Storage.[6]

In the design of these configurations, only the interface description of the embedded components (e.g. Figure 4) is relevant as the inner structure can be neglected. Therefore, we specify the required structure and communication links for each discrete state and assign the BC component in the appropriate state to it. E.g., the BC component instance in state Reference has been assigned to the state AllAvailable where all signals are available. Therefore, a state switch in the Monitor statechart *implies* a state change in the BC statechart. Simple consistency checks ensure that the verified real-time behavior still holds in spite of embedding hybrid components [4].

3. TOOL INTEGRATION

Figure 6 illustrates the way these semantic concepts are used to achieve the desired tool integration between CAMeL and Fujaba: Both tools export hybrid components, which are then integrated into a common hierarchical model.

The tools' output is stored using an exchange format for the description of hybrid components. It contains a high-level interface description, consisting of the hybrid interface statechart (incl. signal flow information), a behavioral description at the source code level and a tool-specific section that allows subsequent modifications using the respective originating tool. As it is the de facto standard for mechatronic systems, C/C++ is used for the low level descriptions. The integration itself is carried out using only the interface descriptions, considering the individual components as black boxes.

3.1 CAMeL

The CAE Tool CAMeL is used for modeling the dynamics of physical systems and for specifying feedback controllers.

[6]Note that the interaction with the Registry is not shown.

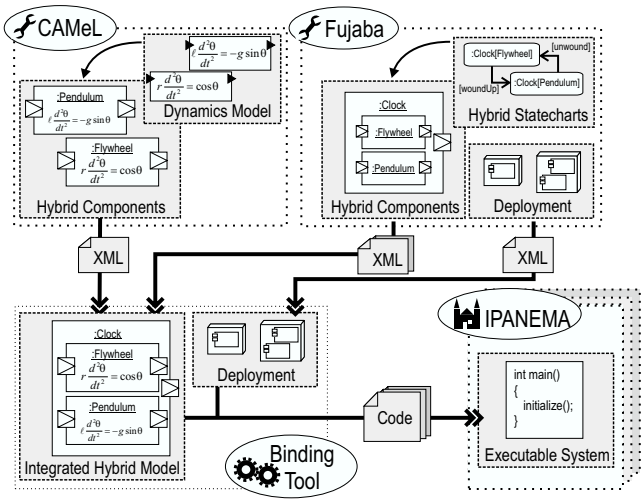

Figure 6: Tool Integration Overview

C++ code is generated from the designed block diagrams. It is executed or simulated within the run-time framework IPANEMA (see Section 3.3). In order to achieve the projected integration, controller block hierarchies may be exported as hybrid components, consisting of the required interface description and generated C++ code implementing the system's differential equations.[7] The required extensions are currently being implemented within the scope of a bachelor's thesis.

3.2 Fujaba

Fujaba currently offers a wide range of UML-based diagrams for the complete specification of (real-time) software. Of particular interest in the current context are component diagrams, deployment diagrams, and Real-Time Statecharts. Discrete components already play a prominent role in the composition and verification of systems, and for reuse based on design patterns. From the specification, code for the Java Real-Time platform may then be generated.

Within the scope of a student research project, the tool suite is now being adapted to incorporate the proposed hybrid concepts by introducing support for hybrid components, ports, and statecharts. Building closely on the existing conceptual framework, the code generation is undergoing a massive rewrite in order to allow the generation of the required C++ code.

3.3 Binding Tool & Run-Time Framework

Though the exchange format provides an integrated model of the complete component hierarchy, an additional step is required to carry this conceptual integration to the execution level. The binding tool determines the correct evaluation order from the signal flow information and then correctly interconnects the individual components.

As we do not consider the outlined integration approach as limited to Fujaba and CAMeL only, the binding tool (see Figure 6) under development operates at the interface specification level and merely binds the code generated by other tools without taking their internal model into account.

[7]Note that the interface statechart of a continuous component consists of just one discrete state.

This means that any tool can provide hybrid components for the binding tool, if it uses the exchange format and provides the C++ code itself. In order to ensure a correct integration, the generated code fragments need to adhere to the requirements set by a common run-time platform.

This platform is IPANEMA 2, a new run-time framework that will introduce reconfiguration and support for C++ into the existing C-based IPANEMA framework [7]. It aims to provide a common environment for the simulation of reconfigurable mechatronic systems, both in pure software and with hardware-in-the-loop (HIL) execution. Using the deployment information it receives from Fujaba, the binding tool is capable of setting up such a simulation and assigns the individual components to their respective nodes.

4. RELATED WORK

Support for the integration of continuous behavior is currently not provided within standard UML. The need for such support is underlined by the OMG request for a proposal of UML for Systems Engineering [10].

In the hybrid extensions HyROOM [14] and HyCharts [6], two hybrid extensions of ROOM [13], complex architectures are specified in a way similar to ROOM, but the behavior is specified through statecharts whose states are associated with continuous models. Their approach, however, is restricted to a non-modular style of reconfiguration and does therefore, unlike the outlined approach, not support reconfiguration for complex, hierarchical systems.

A couple of modeling languages have been proposed to support the design of hybrid systems (e.g. [1, 15]). Most of these approaches provide models, like linear hybrid automata [1], that enable the use of efficient formal analysis methods but lack methods for structured, modular design and reconfiguration.

The de facto industry standard for modeling hybrid systems is MATLAB/Simulink and Stateflow.[8] Modeling reconfiguration is achieved by adding discrete blocks, whose behavior is specified by statecharts, to the block diagrams. Thus, continuous and discrete behavior are separated and not integrated as required for modeling the reconfiguration of complex systems.

5. CONCLUSION AND FUTURE WORK

The presented approach is only a first step towards a complete integration between mechatronics and software engineering. It enables the seamless integration of CAE artifacts into UML in a modular fashion. Thus, the specific complexity and propblems of the different disciplines such as the stability of the control behavior or the correct real-time coordination of the components can to some extent be addressed separately.

In the future, we plan to further strengthen and extend this integration. While currently the real-time processing and the quasi-continuous digital control are combined in a rather static manner, we plan to extend our approach to also cover more dynamic reconfiguration scenarios as well as compositional adaptation [8].

REFERENCES

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(3-34), 1995.

[2] S. Burmester and H. Giese. The Fujaba Real-Time Statechart PlugIn. In *Proc. of the Fujaba Days 2003, Kassel, Germany*, October 2003.

[3] S. Burmester, H. Giese, and O. Oberschelp. Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In *Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal*. IEEE, August 2004.

[4] H. Giese, S. Burmester, W. Schäfer, and O. Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*. ACM, November 2004. (accepted).

[5] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM press, September 2003.

[6] R. Grosu, T. Stauner, and M. Broy. A modular visual model for hybrid systems. In *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98), LNCS 1486*. Springer-Verlag, 1998.

[7] U. Honekamp. *IPANEMA - Verteilte Echtzeit-Informationsverarbeitung in mechatronischen Systemen*. PhD thesis, University of Paderborn, 1998.

[8] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng. Composing Adaptive Software. *IEEE Computer*, 37(7), July 2004.

[9] O. Oberschelp, A. Gambuzza, S. Burmester, and H. Giese. Modular Generation and Simulation of Mechatronic Systems. In *Proc. of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI), Orlando, USA*, July 2004.

[10] Object Management Group. *UML for System Engineering Request for Proposal, ad/03-03-41*, March 2003.

[11] K. Ogata. *Modern Control Engineering*. Prentice Hall, 2002.

[12] OMG. UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02, September 2002.

[13] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, Inc., 1994.

[14] T. Stauner, A. Pretschner, and I. Péter. Approaching a Discrete-Continuous UML: Tool Support and Formalization. In *Proc. UML'2001 workshop on Practical UML-Based Rigorous Development Methods – Countering or Integrating the eXtremists*, pages 242–257, Toronto, Canada, October 2001.

[15] R. Wieting. Hybrid high-level nets. In *Proceedings of the 1996 Winter Simulation Conference*, pages 848–855, Coronado, CA, USA, 1996.

[8] www.mathworks.com

Modifications of the Fujaba Statechart Interpreter for Multiagent-Based Discrete Event Simulation

Nicolas Knaak
University of Hamburg
Department for Informatics
Vogt-Koelln-Strasse 30
22527 Hamburg, Germany
knaak@informatik.uni-hamburg.de

Keywords
UML Statecharts, Fujaba, Multiagent-Based Simulation

ABSTRACT
In this contribution the Java framework FAMOS for multi-agent-based discrete event simulation is introduced. The framework supports different methods for behaviour modelling like rule-based descriptions and executable UML statecharts. The statechart interpreter used in the framework is based on the statechart execution algorithm of Fujaba 3. The original algorithm is extended with a correct treatment of inter-level transitions, final states and orthogonal regions with multiple history states according to the UML semantic. Further changes are made to increase its performance in models with many agents.

1. INTRODUCTION
Multiagent-based simulation (MABS) has become an important means for system analysis in several application domains like social science, biology or logistics. MABS is characterized by a microscopic modelling perspective, i.e. the system to be analysed is modelled from the point of view of the participating autonomous, self-interested and goal-directed actors (agents). Compared to related simulation techniques like individual-based simulation [13, pp. 51], object-oriented simulation or process interaction [21, pp. 30][1], MABS places stronger emphasis on modelling the agents' complex behaviours, communications and cooperations, borrowing many concepts from distributed artificial intelligence (see e.g. [23]). According to Klügl MABS can be considered a more general superset of traditional microscopic simulation techniques [13, p. 70].

[1]Actually object oriented programming has its roots in the language Simula[7] that was designed for simulation applications. The Simula-based DEMOS package even allowed the implementation of rather "agent-like" simulation processes.

Today there are numerous software frameworks supporting the development of agent-based models. Most of them, like e.g. the well-known *Swarm* system [3], are tailor-made for building spatially explicit grid-based models where simulation time advances in equidistant steps. However, in domains like transport or logistics that are characterised by complex asynchronous processes, simulations with event-driven time advance might be more appropriate.

2. A FRAMEWORK FOR AGENT-BASED MODELLING AND SIMULATION
Motivated by the needs of a research project on the simulation of courier service logistics [15] the agent-based simulation framework FAMOS (Framework for Agent-based Modelling and Simulation) was developed at the University of Hamburg. FAMOS is based on the discrete event simulator DESMO-J [22], which supports event scheduling, process interaction, activity-oriented and transaction-oriented simulation.[2] FAMOS extends DESMO-J with a framework for spatial modelling with grids, graphs and continuous models [19, 8] as well as constructs for modelling the behaviour of agents [14].

Agents in FAMOS are active entities that communicate explicitly via typed signals. The notion of a signal roughly corresponds to the UML semantics [6], an event in Fujaba or a message in an agent platform like e.g. JADE [2]. Signals are either received from other agents or scheduled at future points in simulation time by the agent itself. This explicit communication mechanism fits discrete event simulation very well and builds the basis for modelling reactive and pro-active (i.e. self-initiated) agent behaviour.

Due to the variety of available architectures for implementing agent behaviour (e.g. [13, pp. 21]) FAMOS agents are associated an exchangeable behaviour object for signal handling (see figure 1). Different agent architectures can be implemented by extending a basic `Behaviour`-class.[3] The current version of FAMOS contains subclasses for simple event- and process-oriented behaviour modelling as well as

[2]DESMO-J (Discrete Event Simulation and Modelling in Java) is a Java port of the Modula II simulation package DESMO [21] that builds on the concepts of Birtwistle's DEMOS.

[3]Similar architectures can be found in some agent platforms like JADE [2] or MadKit [5].

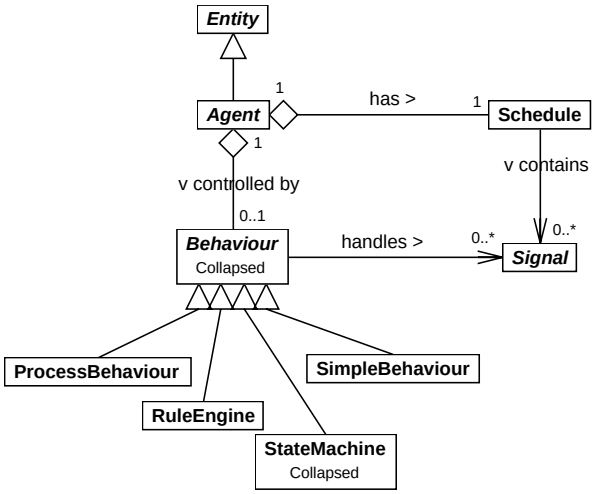

Figure 1: Class diagram showing the structure of an agent in FAMOS.

rule-based behaviour description using the JESS expert system shell [1] and an architecture for deliberate (i.e. dynamically planning) agents [8].

Considering the importance of UML state- and activity-charts for modelling reactive agent behaviour (see e.g. [20, 11]) a framework for executable statecharts is also available. Statecharts can be specified using either an XML-based script or a simple graphical editor.

3. MODIFICATIONS OF THE STATECHART INTERPRETER

The integration of executable statecharts into the FAMOS framework is based on the statechart interpreter of Fujaba 3 [4] developed by Köhler as a part of his diploma thesis [16, ch. 4]. Compared to other approaches the underlying algorithm stands out by its straightforward implementation of the UML semantics and covers most statechart elements. The Fujaba statechart framework rests upon the idea of an "object-oriented state table" [17, p. 4] representing state machines as graphs of state and transition objects. Composite states are realized as "states containing substates" according to the *Composite* pattern [9]. Each composite state can be assigend an initial state with an associated history type (none, shallow or deep). Concurrent states are implemented in an ad-hoc fashion as composite states with multiple initial states.

During execution the interpreter stores the leaf states of the state machine's configuration tree in a list. When an event occurs, all states in the list are sequentially searched for enabled transitions. According to the UML semantic the interpreter searches transitions from inside out, i.e. leaf states try to handle an event first and eventually propagate it to their parent states. The Fujaba statechart interpreter supports entry-, exit and do-activities of states as well as guards and effects of transitions.

However, to the author's understanding of the code (based on a static analysis) there are some deviations from the UML

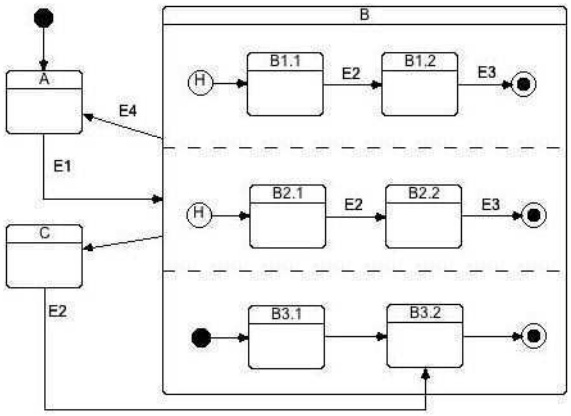

Figure 2: An example concurrent statechart with multiple history states and explicit entry through an inter level transition.

semantics:

- Since the history type can only be set for the first orthogonal region of a composite state, concurrent states with multiple histories (see figure 2) are not allowed.
- Though composite states might be left explicitly via inter-level transitions (i. e. outgoing transitions starting from one of the substates), an explicit entry (as from state C to state B3.2 in figure 2) seems not possible. In this case the interpreter enters the correct state but does not execute entry activities of parent states or enter parallel orthogonal regions.
- There is no difference between self-transitions (i.e. transitions with the same source and destination) and static reactions (i.e. reactions to events that do not include a state change). Actually, self-transitions in a Fujaba statechart are treated like static reactions since entry- and exit-activities of the assigned state are not executed.
- The use of final states and default transitions does not conform to the standard UML. A composite state should be left via outgoing default transitions when all orthogonal regions have reached a final state. However, in Fujaba an event triggering the default transition seems to be sent immediately after executing the do-activity.

The FAMOS statechart framework is a modified and extended re-implementation of the Fujaba statechart interpreter and framework (see [17, p. 5]). Focus is put on an exact implementation of the UML statechart semantic as well as on increased performance, which is crucial in models with many agents. Conformance to the standard UML is preferred since FAMOS will be used in courses for students already familiar with basic UML constructs. Inter-level transitions are sometimes deemed to be problematic because they prohibit the modular composition of statecharts (see e.g. [10]), but we nevertheless regard them as a convenient modelling construct. Due to the complexity of

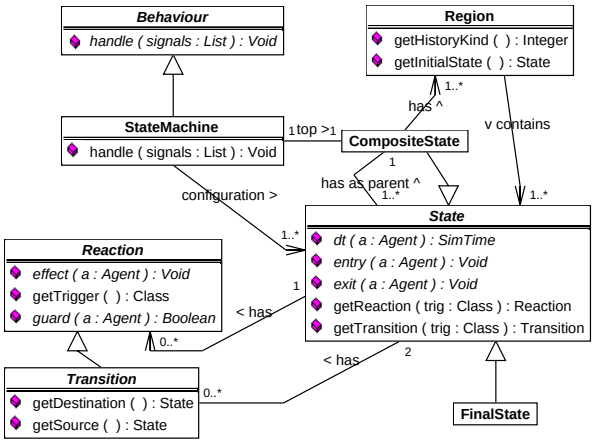

Figure 3: The FAMOS statechart framework.

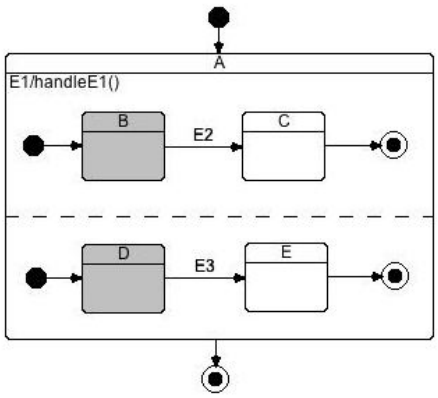

Figure 4: An example for the execution of reactions and transitions.

agent behaviours, concurrent states with orthogonal regions might also be relevant in MABS.

In the FAMOS statechart framework (see figure 3) orthogonal regions are modelled explicitly as objects and each composite state is associated a list of regions as containers for its substates. When a composite state is entered, the statechart interpreter determines the entry type (default, history or inter-level transition) of every region and enters the respective substates. For each active composite state it keeps track of the number of orthogonal regions that have reached a final state. After all regions are finished, a signal triggering the state's outgoing default transitions is sent. When the state is left, the current state of each region with a history is stored in a hashtable.[4]

Other than Fujaba, the FAMOS statechart interpreter separately handles transitions and static reactions. The statechart framework contains a class for reactions (aggregating a trigger attribute, a guard and an effect method) as well as a derived class for transitions with additional attributes for source and destination states. According to the UML semantic a state's reactions are prioritised before its outgoing transitions. Furthermore it is possible to differentiate between self-transitions that leave and re-enter a state and reactions with no state change at all.

A problem might occur in the statechart from figure 4 in configuration $[B, D]$. On handling an event $E1$ the interpreter first searches state B for a transition (without success) and then proceeds to its parent state A executing the assigned reaction. Since D is still in the configuration, the interpreter searches this state next. D can not handle $E1$ and therefore propagates the event to A, thus executing the reaction again. As a workaround for this unwanted behaviour the interpreter ensures that every reaction is only executed once for every event. Generally a breadth first transition search (i.e. *all* leaf states are searched before ascending to their parents) might be a better solution.

The last modification regarding the semantic of UML statecharts is concerned with incoming inter-level transitions. When executing the transition from state C to state $B3.2$ in figure 2 the Fujaba statechart interpreter would to the author's impression directly insert the destination state $B3.2$ into the configuration. Entry activities of the parent state B are not executed and parallel states (e.g. $B1.1$ and $B2.1$) are not entered at all. The FAMOS interpreter determines the highest parent state entered by the transition instead (B in the example) and enters it recursively. In the example, $B3.2$ is entered explicitly while the parallel regions are entered via their history states.

Efficiency is an important criterion when using executable statecharts in MABS. Agent-based models often contain a large number of agents that must be executed in parallel on a single processor. To enhance the statechart interpreter's efficiency the following changes were made:

- The state table is a static attribute of the agent, i.e. it only exists once at runtime and is used by all agents sharing the same behaviour. As a payoff the `State` objects' internal attributes have to be externalized. An alternative proposed in [17, p. 8] is the use of the *Flyweight* pattern [9].
- State and transition actions are not realized using reflection, that still lacks performance in Java. Instead corresponding abstract methods (e.g. `entry()`) of the `State` or `Transition` class must be implemented (see [16, p. 62]).
- To avoid the overhead of thread handling, agents in FAMOS do not run in a thread like active objects[5] in Fujaba. The implementation of a state's potentially time consuming do-activity is no longer possible which can be considered a minor drawback in the context of discrete event simulation.
- The Fujaba statechart interpreter re-computes information about the statechart structure in every step though the values remain constant. Examples are the

[4]Different from Fujaba the FAMOS statechart interpreter only supports shallow history states so far.

[5]An active object is an object that runs in its own thread of control and is capable of asynchronous event handling [18].

depth of states or the highest state left by a transition. In FAMOS these values are only determined once when the statechart is built.

4. APPLICATION OF STATE-BASED MODELLING TO MABS

The FAMOS statechart framework and modelling tools were applied in a research project on the simulation of alternative logistic strategies for courier services [15]. Couriers and radio operators of a courier service were modelled as reactive state-based agents with additional deliberation capabilities e.g. for tour planning. The performance of the statechart interpreter was sufficient to execute models of about 100 concurrently active agents moving on a detailled road network in reasonable time. On the one hand we regarded visual modelling with UML statecharts as useful for analysing and debugging agents' behaviours. On the other hand some important facets of agent-oriented modelling like deliberation or scheduling of concurrent tasks can not be expressed adequately. Furthermore statecharts for complex agents often grow too large to be handled conveniently. The latter aspect might be improved by the enhanced modularity of UML 2.0 statecharts that allow explicit entry and exit points of submachine states (see e.g. [12, p. 311]).

5. CONCLUSION AND OUTLOOK

The FAMOS framework for MABS uses a modified version of the Fujaba 3 statechart interpreter for modelling agent behaviour with executable statecharts. The original interpreter was tuned for performance, and the handling of inter-level transitions, final states and orthogonal regions with multiple histories was corrected. While state-based modelling has been successfully applied in simulations of courier service logistics, a detailled performance comparison between the statechart interpreters of FAMOS and Fujaba is still missing. Moreover it should be examined if the modifications described above can be fruitfully applied to the statechart frameworks of Fujaba 3 and 4. Besides this we are currently exploring the usefulness of the new UML 2.0 notation for discrete event modelling and simulation in general.

6. REFERENCES

[1] http://herzberg.ca.sandia.gov/jess.

[2] http://jade.tilab.com.

[3] http://wiki.swarm.org.

[4] http://www.fujaba.de.

[5] http://www.madkit.org.

[6] http://www.omg.org/cgi-bin/doc?formal/03-03-01.pdf.

[7] G.M. Birtwistle. *DEMOS, a System for Discrete Event Modelling on Simula*. Macmillan, London, 1979.

[8] R. Czogalla and B. Matzen. Agentenbasierte Simulation von Personenbewegungen in kontinuierlichem Raum. Diploma thesis, University of Hamburg, Department for Informatics, 2003.

[9] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1998.

[10] M. Glinz. Statecharts for Requirements Specification - As Simple as Possible, as Rich as Needed. In *Proceedings of the ICSE 2002 Workshop on Scenarios and State Machines: Models, Algorithms and Tools*, Orlando, Florida.

[11] M.L. Griss, S. Fonseca, D. Cowan, and R. Kessler. Using UML State Machines Models for more Precise and Flexible JADE Agent Behaviours. In *AAMAS AOSE Workshop*, Bologna, Italy, July 2002.

[12] M. Jeckle et al. *UML 2 glasklar*. Hanser, München, 2004.

[13] F. Klügl. *Multiagentensimulation - Konzepte, Werkzeuge, Anwendung*. Addison-Wesley, 2001.

[14] N. Knaak. Konzepte der agentenbasierten Simulation und ihre Umsetzung im Rahmen des Simulationsframeworks DESMO-J. Diploma thesis, University of Hamburg, Department for Informatics, 2002.

[15] N. Knaak, R. Meyer, and B. Page. Agent Based Simulation of Sustainable Logistic Concepts for Large City Courier Services. In *EnviroInfo 2003 - 17th International Conference Informatics for Environmental Protection*, pages 318 – 325, Cottbus, September 2003.

[16] H.-J. Köhler. Code-Generierung für UML Kollaborations-, Sequenz- und Statechart-Diagramme. Diploma thesis, University of Paderborn, 1999.

[17] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Using UML as a Visual Programming Language. Technical Report tr-ri-99-205, University of Paderborn, Paderborn, Germany, August 1999.

[18] R.G. Lavender and C.D. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. *Proc.Pattern Languages of Programs,*, 1995.

[19] R. Meyer. *Agenten in Raum und Zeit: Agentenbasierte Simulation mit expliziter Raumrepräsentation (in preparation)*. Dissertation, University of Hamburg, Department for Informatics, 2004.

[20] C. Öchslein, F. Klügl, R. Herrler, and F. Puppe. UML for Behaviour-Oriented Multi-Agent Simulations. In B. Dunin-Keplicz and E. Nawarecki, editors, *Proceedings of the 2nd International Workshop of Central and Eastern Europe Multi-Agent Systems*, 2001.

[21] B. Page. *Diskrete Simulation - Eine Einführung mit Modula 2*. Springer, Berlin, 1991.

[22] B. Page, T. Lechler, and S. Claassen. *Objektorientierte Simulation in Java mit dem Framework DESMO-J*. Libri Book on Demand, Hamburg, 2000.

[23] G. Weiss, editor. *Multi-Agent-Systems*. MIT Press, Cambridge (MA), 1999.

Component Templates for Dependable Real-Time Systems*

Matthias Tichy, Basil Becker, and Holger Giese
Software Engineering Group, University of Paderborn, Warburger Str. 100, Paderborn, Germany
[mtt|basilb|hg]@uni-paderborn.de

ABSTRACT

A general trend towards more complex technical systems can be observed which results in an increasing demand for methods and tools to develop dependable, high quality software for embedded systems. The UML in principle provides the essential concepts which are required to model such complex, safety-critical software systems. In this paper, we describe a component template plugin for the Fujaba Real-Time Tool Suite which has been especially tailored to support fault-tolerance templates such as triple modular redundancy. We report about the underlying concepts and the application of the plugin by means of an example.

1. INTRODUCTION

Due to the trend that more and more ambitious and complex technical systems are built today, an increasing demand for dependable, high quality software can be observed. This trend is characterized in [11] by very complex, highly integrated systems of systems with subsystems that must have a great degree of autonomy and, thus, are very demanding w.r.t. safety analysis. The New Railway Technology (RailCab) project[1] tackled by our efforts for the Fujaba Real-Time Tool Suite is one very extreme example for such complex systems of systems with very demanding safety requirements.

In such engineering projects, most often the involved engineers are not safety experts and, thus, sophisticated, application specific fault-tolerance techniques can often not be realized. Systematic fault-tolerance approaches such as triple modular redundancy (TMR), n-version programming, hot stand-by, etc. [14] can in contrast be employed by non experts.

However, in practice the additional complexity and pitfalls during their implementation are often a hindrance to finally achieving the intended improved dependability. We therefore propose to support the design of fault-tolerant systems by means of templates and automate the code generation for the additional logic. The templates permit to *reuse* well analyzed and understood solutions for systematic fault-tolerance and therefore minimize the risk that inadequate and error prone ad hoc solution are invented. The automatic generation of the glue logic can further exclude coding faults and therefore exclude that the additional complexity which results from the application of systematic fault-tolerance approaches themselves deteriorates the dependability of the resulting system in practice.

The UML as an object-oriented technology is one candidate to handle these safety-critical systems with software and overwhelming complexity. However, the current and forthcoming UML versions do not directly support the design of fault-tolerant designs for safety-critical system development. The presented approach tries to narrow the described gap between safety-critical system development and available UML techniques. As there is little value in proposing extensions to UML if they are not accepted by the community and tool vendors (cf. [11]), we instead propose to use only a properly defined subset of the UML 2.0 [12] component and deployment diagrams and templates for fault-tolerance to ease the task of integrating systematic fault-tolerance techniques into a UML design.

After reviewing related work in Section 2, we present our approach for component templates for fault-tolerance in Section 3. The provided tool support and application of the Fujaba Real-Time Suite Plugin are described in Section 4. We finally present some conclusions and give an outlook to planned future work.

2. RELATED WORK

Templates are a standard approach used in many different application areas. Constructs like C++ templates [15] and Java Generics [16] offer templates for programming languages. In the modeling domain templates are available in a number of different contexts.

For multimedia artifacts Cybulski [6] presents the *Template* pattern. The Template pattern is a solution to the "[...] need to produce a collection of composite artifacts similar in structure and contents." The Template pattern provides support for (1) the structural specification of a collection of composed artifacts and (2) the instantiation of the template by replacing template artifacts by concrete artifacts. Although destined for the use in multimedia applications, the Template pattern can be tailored to the use in real-time component-based applications. Our component template plugin implements an extended variant of this pattern.

The UML [12, p.541 ff.] includes a Template package. This Template packages provides Metaclasses, which allow the specification of TemplateElements which have a number of TemplateParameters. A TemplateBinding "specifies the substitution of actual parameters for the formal parameters

*This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

[1]http://www-nbp.upb.de

of the template". The general UML template approach allows template applications with meaningless template bindings. While such an approach is possible for modeling standard software, we target the domain of safety-critical embedded systems. Thus, a more strict template mechanism is required. The presented template mechanism is similar to the general one in the UML, but includes special support for real-time component templates in contrast to the general UML template mechanism. Here special support means, that e.g. if a component has been used in the application of a template, only ports of this component and not arbitrary ones can be used.

3. COMPONENT TEMPLATES

In the following, we will present the Fujaba real-time component diagrams. Thereafter, we will show how component templates are specified and how they can be applied.

The Fujaba real-time component diagrams [5, 8] are based on concepts originally proposed in ROOM and UML 2.0 [12, 13]. They are used for the specification of system structure. Component diagrams specify components and their interaction in form of connectors. We differentiate component types and their instances during runtime. Connectors model the communication between different components via ports and interfaces. Ports are distinct interaction points between components and are typed by provided and required interfaces. Behavior of components is specified by real-time statecharts [3]. As this paper deals with structural templates, we will, in the following, not consider the behavioral aspect of components and component templates. As example, we will use the triple-modular-redundancy (TMR) fault tolerance technique.

3.1 Template specification

In a first step, the structure of the fault-tolerance technique must be specified. This is done by creating a component template specification. This specification is merely a standard real-time component diagram.

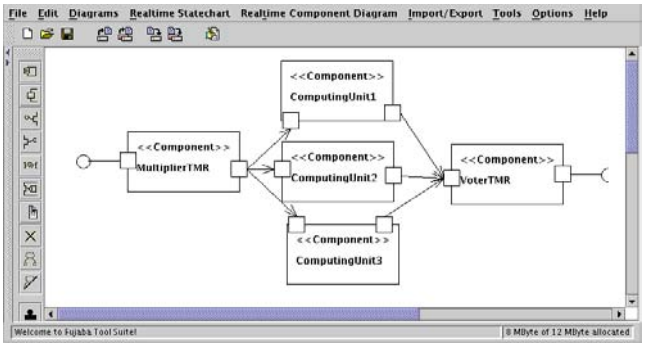

Figure 1: Component template specification for TMR

Figure 1 shows the component template specification for the triple modular redundancy fault tolerance technique. A triple modular redundancy system uses a multiplier component which triples the input received and forwards it to the three services ComputingUnit1 . . . 3, which actually perform the computation. The voter compares the different results and chooses the result which at least two of the components returned. Thus, a triple modular redundancy system can tolerate one crashed or malfunctioning service. The ports and interfaces attached to the left of the MultiplierTMR component and right of the VoterTMR component are not part of the fault tolerance pattern but are important for connecting the using and used components during application of the component template.

3.2 Application

The specified fault tolerance component templates are later used to build more robust applications by applying them to the component structure of a system. First, an appropriate component template is selected and added to the component structure. Then, the different parts of the component template must be replaced by the actual implemented parts. The different parts here are components, ports, and interfaces. Thus, a mapping must be defined by the user between the components of the component template and implemented components. Thereafter, the ports of this template component and the ports of the implemented component are mapped. Finally, the interfaces attached to the ports are mapped. The mapping of interfaces must respect the type of the interfaces, e.g. *required* interfaces of the template component must be mapped to *required* interfaces of the implemented component. In addition the interface of the implemented component must be a subclass of the interface of the template component. This constraint offers support for more application specific templates where consistency between interface types is required. However, this constraint can be relaxed for broader usage of the component template.

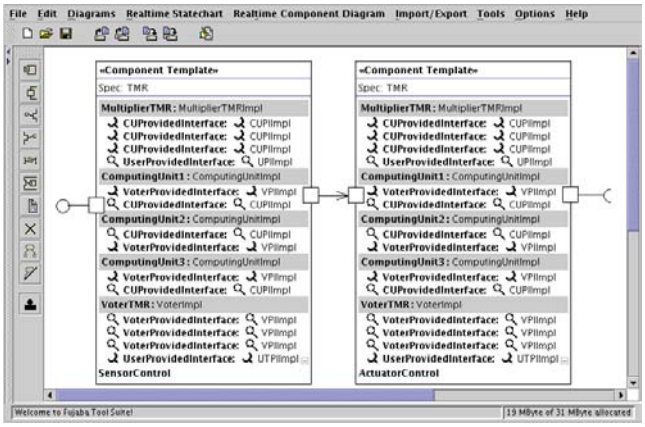

Figure 2: Application of the TMR template

Figure 2 shows the application of the TMR component template. The TMR template is applied for the sensor control software as well as for the actuator control software as shown on the bottom of each component in Figure 2. The different parts of the TMR template are already mapped to implemented components, ports, and interfaces.

After all parts of the template are mapped to implemented parts, the template in the component diagram is replaced by the template structure (cf. Figure 1) using the mapped implemented parts. This replacement, thus, makes the fault tolerance techniques explicit in the design of the system. After this step, the standard code generation of the Fujaba Real-Time Tool Suite can be employed to synthesize the source code for the fault-tolerance enhanced system.

3.3 Multistage arrangement

A fault tolerance template like TMR can be employed multiple times in the component structure of an embedded system. A naive usage of this template would result in a situation where three redundant components are connected to three other redundant components by single voter and multiplier components. Thus, the redundancy gained by the application of the TMR template is defeated by the single-point-of-failure voter and multiplier components.

A better approach is the usage of a multistage arrangement. A multistage arrangement uses redundant voter and multiplier components in contrast to the mentioned single voter and multiplier approaches. Thus, a transformation of multiple applications of TMR or other templates to a multistage arrangement is important for the fault-tolerance of the system. Using the Story-Pattern language [7, 10] of Fujaba, it is possible to specify an accordant transformation from a multiple application of TMR to a multistage arrangement.

After this description of component templates, we present in the next section the tool support offered by the component template plugin.

4. TOOL SUPPORT

We have developed a Fujaba plugin that provides tool support for the mentioned component template specifications and the mappings between template components, ports and interfaces and implemented ones. In the following, we highlight the plugin dependencies, the meta-model extension, and special mapping support.

Plugin structure

The developed RealtimeComponentTemplate plugin (RCT plugin) depends on two other plugins developed at the University of Paderborn. First it depends on the RealtimeComponent (cf. [5]) plugin. We use the RealtimeComponent plugin's component diagrams for the specification of component templates. As the RealtimeComponent plugin depends on the RealtimeStatechart plugin [3] our RCT plugin depends on it, too.

Meta-model extension

How are the mappings between components, interfaces and ports realized? At first sight one might think that this won't be a problem, because all used classes are defined in one plugin - the RealtimeComponent plugin. The easiest way would be to define one-to-many self-associations between the component, port, and interface classes. But then we would have to change the RealtimeComponent plugin which we wanted to avoid. Fortunately, Fujaba provides the ASG mechanism [4] to avoid these problems. In the following paragraphs, we will restrict our explanations to the mapping of components. The mapping of ports and interfaces is done analogously.

Each mapping between components is represented by an instance of RTCompMapping. This mapping has two references to an extension of ASGElementRef; one for the incoming and one for the outgoing mapping. These extensions of ASGElementRef have a one-to-one ASG reference to the Component class. As we need one-to-many associations between Component and RTCompMapping (e.g. one component from a given specification is used in more than one template application), however ASG only provides one-to-one associations, we implemented the one-to-many association between RTCompMapping and the ASGElementRef extensions. The ASGElementRef extensions have the role of a proxy for the Component class. This means we displaced the one-to-many association needed in our plugin. As mentioned earlier the interface and port mapping is implemented in the same way.

A few words on the hierarchy of the different mapping types: Each template has a set of component mappings to implemented components. Since every port is owned by exactly one component, the port mappings are stored in each component mapping. The same argumentation applies for interface mappings.

Mapping support

In the following, we present the way in which the plugin supports the mapping specification. Tool usability[2] is a major factor for the acceptance of an approach and its supporting tool. Therefore, we explicitly emphasized usability especially for the mapping specification.

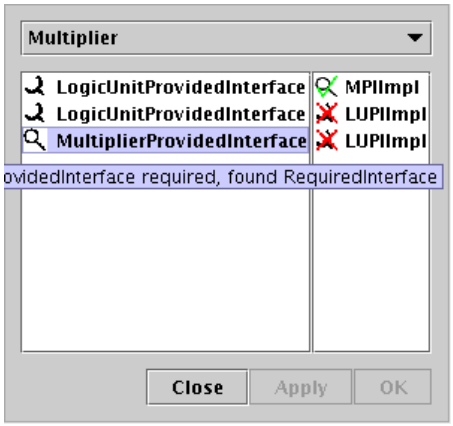

Figure 3: Interface mapping dialog

As mentioned, interface mapping is subject to constraints regarding the type of the interfaces (required/provided) and the subclass relation between template interfaces and implemented interfaces. To improve the usability of this interface mapping the interface mapping dialog produces immediate, easy to understand feedback. This means every time you select two interfaces you want to map, the plugin checks whether this mapping will be correct or not. A mapping of interfaces is correct if both interfaces are of the same type (provided or required interface) and the interface declared in the implementation is a subclass of the interface in the specification. This direct feedback is realized either by a red crossed out or by a green checked off interface icon (cf. Figure 3). If the mapping is incorrect the tooltip of the interface in the implementation gives a short explanation why the mapping is wrong.

[2]To introduce our understanding of usability we will first give a short definition of it found in [1]: "Usability is the ease with which a user can learn to operate, prepare inputs for and interpret outputs of a system or component."
The expression usability is used in different contexts. In our context, i.e. software development, usability is often called *software ergonomics*. Ergonomics is the conformity of technology (i.e. software) to human psychophysical capabilities. For further details on ergonomic and usable software see [2].

Concerning the order of the different mappings, the standard order would be mapping components, then ports, and then interfaces. We provide a small shortcut for this mapping based on the fact that a interface can only be connected to exactly one port, i.e. if you know the interface, you know the corresponding port. The plugin exploits this property and allows you to map the interfaces without mapping the ports. Every time you map an interface, the corresponding port mapping is automatically determined.

5. CONCLUSION AND FUTURE WORK

We presented an approach for specification of dependable, component-based, embedded systems. We aim for improving the fault tolerance of distributed systems by the application of fault tolerance component templates. Tool support for specification and application of component templates has been developed including special care for the usability of the mapping.

Currently, the component templates only cover the structural parts of fault tolerance techniques. In the future, we will also consider the glue logic resulting of the behavior of voter and multiplier components. We believe, that the behavior of these two component types can in principle be synthesized based on the behavior of the other components.

In the future, we will consider the deployment issues of fault tolerance templates. Fault tolerance templates typically employ redundancy to enhance fault tolerance. If the redundant components of a fault tolerance template are deployed to the same host, there is no fault tolerance w.r.t. failures of this host. Thus, appropriate deployment constraints must be specified for fault tolerance templates. These constraints would specify that each of the redundant components must be deployed to different hosts.

In [9], an approach for a compositional hazard analysis of component-based systems is presented. The knowledge about the employed fault tolerance provided by the templates could be exploited in the hazard analysis.

Currently, the user can use arbitrary components in the application of the fault tolerance template. Fault tolerance techniques like n-version programming explicitly request heterogeneous components in order to tolerate systematic implementation errors. Adding appropriate constraints to component templates would ease the use of component templates for heterogeneous fault tolerance techniques.

REFERENCES

[1] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers, New York, 1990.

[2] A. Brenneke, R. Keil-Slawik, and W. Roth. Designorientierung und Designpraxis - Entwicklung und Einsatz von konstruktiven Gestaltungskriterien. In U. Arend, E. Eberleh, and K. Pitschke, editors, *Software-Ergonomie '99 Design von Informationswelten*, pages 43–52. B. G. Teubner Stuttgart, 1999.

[3] S. Burmester and H. Giese. The Fujaba Real-Time Statechart PlugIn. In *Proc. of the Fujaba Days 2003, Kassel, Germany*, October 2003.

[4] S. Burmester, H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004. (accepted).

[5] S. Burmester, M. Tichy, and H. Giese. Modeling Reconfigurable Mechatronic Systems with Mechatronic UML. In *Proc. of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden*, June 2004.

[6] J. L. Cybulski and T. Linden. Composing Multimedia Artifacts for Reuse. In *Proc. of the 1998 Pattern Languages of Programs Conference, Monticello, Illinois, USA*, August 1998.

[7] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, November 1998.

[8] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[9] H. Giese, M. Tichy, and D. Schilling. Compositional Hazard Analysis of UML Components and Deployment Models. In *Proc. of the 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP), Potsdam, Germany*, Lecture Notes in Computer Science. Springer Verlag, September 2004. (to appear).

[10] H. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE), Limerick, Ireland*, pages 241–251. ACM Press, 2000.

[11] J. A. McDermid. Trends in Systems Safety: A European View? In P. Lindsay, editor, *Seventh Australian Workshop on Industrial Experience with Safety Critical Systems and Software*, volume 15 of *Conferences in Research and Practice in Information Technology*, pages 3–8, Adelaide, Australia, 2003. ACS.

[12] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. Document ptc/03-08-02.

[13] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, Inc., 1994.

[14] N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[15] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991. Second Edition.

[16] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the java programming language. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296. ACM Press, 2004.

Visualizing differences of UML diagrams with Fujaba

[Position Paper]

Jörg Niere
Software Engineering Group
Hölderlinstr. 3
D-57068 Siegen, Germany
joerg.niere@uni-siegen.de

1. MOTIVATION

Software Configuration Management (SCM) is indispensable when developing large software systems. There exists well established tool support such as the popular open source tool CVS or several commercial systems, e.g. Microsoft Visual SourceSafe. Common to those tools is that the management is based on text files and therefore mostly used in the late phases of the software development process, i.e., the implementation phase.

In early phases of a software development process such as the requirements or design phase, engineers use CASE tools and more precise UML modeling tools. Today, CASE tools usually provide a set of different diagrams and use an abstract syntax graph representation, namely a meta-model. Persistently storing an actual syntax graph representing of a set of diagrams in text files is a possible solution, but managing those persistent text files with a textual SCM tool is not feasible [7, 5]. Suppose the situation, that you load a set of diagrams and save the diagrams without changes. If the sequence of diagram items has been changed, e.g. depending on the internally used container classes, the SCM tool will say that the file as changed although the diagrams have not.

CASE tools need their own SCM tool working on the internal abstract syntax graph representation [6] in order to prevent problems such as sequence changes that have no semantical meaning for the diagrams. For example the Rose tool [4] comes along with an SCM tool, which supports calculating differences between two versions of a set of diagrams. The Rose Model Integrator uses the internal unique identifiers to compare diagram items in different versions and represents the differences in a kind of internal abstract syntax tree view. The engineer has to know for example that the class *MethodDecl* is the internal representation of a *method* in a class diagram, or class *RelEnd* represents roles of associations. Especially when there are a large number of differences, the actual context of a certain difference is not always clear to the engineer.

The Fujaba CASE tool also supports versioning of diagrams [8]. Fujaba's approach is to log all edit operations; so-called changed-based versioning [5]. Calculating differences between two versions means to replay the edit operations starting from the common original diagram and using unique identifiers to compare diagram items in the different versions. Fujaba displays difference information about internal syntax graph classes similar to the Rose Model Integrator.

As the previous examples show, the inner tool SCM support is already understood for CASE tools, although the representation of the difference information is inadequate. This position paper sketches our approach of visualizing differences of UML diagrams within the diagrams themselves. We call those diagrams difference diagrams. Although we use Fujaba as tool to display difference diagrams, our approach is independent from a certain internal representation of diagrams, because the calculation algorithm takes two XMI documents as input and does not depend on unique identifiers. Those XMI files may be produced any CASE tool.

2. DIFFERENCE VISUALIZATION

Our approach takes two XMI documents as input for the difference calculation algorithm and the algorithm produces a unified XMI document including difference information. To represent the difference information within the unified XMI document we use the XML extension mechanism. The advantage is that the unified document can also be understood by other CASE tools having a standard XMI import facility where our specific extension will have no effect. Currently the difference calculation algorithm handles class diagrams only and the visualization is under construction.

Figure 1 shows two different class diagrams. Each diagram represents a subset of the internal structure of HTML documents containing advanced features such as forms, lists and combo boxes. Both class diagrams have been produced independently by different development teams during a practical software engineering course at the university. The class diagram on the left hand side contains hard restricted relations between the different elements such as forms only have one combo box element and one list. The class diagram on the right hand side uses the composite design pattern [2] where an actual HTML document can contain a number of forms that can contain a number of lists and combo boxes. In addition, a combo box is subtype of a list, because combo boxes are also lists but showing only the selected item and not the whole list. A design such as on the right hand side in Figure 1 can be found in many graphical user interface libraries, e.g. the Java Swing library. In the following we call the class diagram on the left hand side in Figure 1 sim-

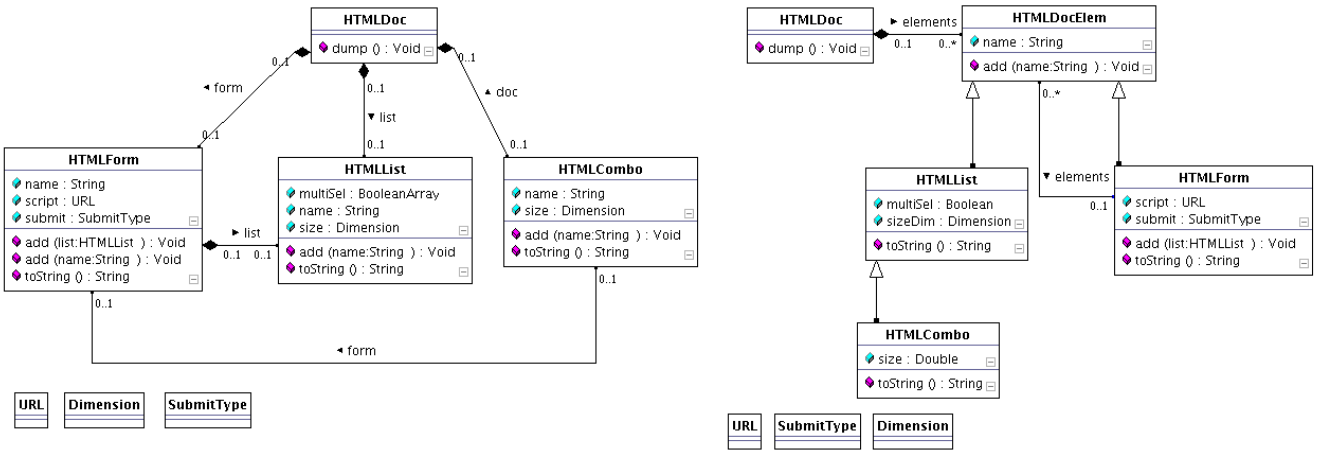

Figure 1: Two class diagram variants of an internal HTML document representation.

ply the left class diagram and the other one the right class diagram.

After three weeks design phase, the different development teams must present and defend their design solutions and all teams have to agree on a common design as basis for further work. In general the common design includes parts from more than one original diagram and the question of how such an operation is supported arises. One solution is to create a unified class diagram containing all classes and relationships in such a way that all elements of both diagrams are contained in the unified diagram. The result is comparable with the union operation presented by Selonen [9]. Consequently you get a unified diagram where the original two diagrams lie next to each other, similar to Figure 1. The similarities in both diagrams such as combo boxes, lists, forms or documents have been completely ignored.

Our difference calculation algorithm tries to identify similar diagram elements automatically, which results in a difference diagram containing all elements of both original diagrams but also glued elements identified as similar. The resulting difference class diagram is shown in Figure 2. In addition, we use Fujaba's internal class diagram layout algorithm. On the one hand layout is an important information, especially when handling large diagrams. On the other hand our approach takes independently developed diagrams with different layout aspects as starting point and thus we currently use standard layout algorithms available in the actual visualization tool and skip the layout information of the diagrams, completely.

Our approach of calculating and visualizing differences bases on the definition of diagram elements. For example, we decided that diagram elements of class diagrams are classes, attribute definitions, method signatures, associations, inheritance relations, notes, etc. Diagram elements consist of attributes and can contain other diagram elements and can also be connected to other diagram elements. Simplified we get something like an internal meta-model for which we can define similarity rules. The difference calculation algorithm applies these rules to the input files after parsing them.

Based on the internal meta-model, we distinguish between three kinds of differences. Most simple are *move differences*, because they indicate a renumbering of elements. For example, a rearrangement of attributes or methods of a class. We visualize move differences by adding a small number button (n-button) to the diagram element. Hence the current implementation automatically sorts attributes and methods by name and all other sequences are ignored, there exists no move difference in Figure 2.

Structural differences indicate that a certain diagram element belongs only to one of the original diagrams, i.e., there exists no similar element in the other diagram. In general, elements contained only in the left class diagram are displayed in red color and elements contained in the right class diagram are displayed in green color. Referring to the screen-shot in Figure 2, the red color becomes light gray and the green color is shown as dark gray.[1] For example, class HTMLDocElem belongs only to the right class diagram and therefore the whole class, i.e., its name and border as well as its attributes and methods, get green color. All other classes have been identified as similar and therefore we visualize them in black color, e.g. HTMLDoc, HTMLForm, HTMLList and HTMLCombo.

The difference calculation algorithm has identified no similar relationships between the classes, i.e., inheritance relationships and associations. Therefore the two elements associations and the two inheritance relations starting from class HTMLDocElem to HTMLForm as well as HTMLList and the inheritance relation between HTMLCombo and HTMLList are shown in green (dark gray) color. All others are shown in red (light gray) color.

Structural differences can also occur within classes as shown in Figure 2. Attributes and methods, which have not been identified as similar are shown in red color if they belong to the left class diagram and shown in green color if they belong to the right class diagram. For example the name attribute

[1] We are working on using dotted line styles and fonts instead of using colors for better readability of gray-scaled screenshots.

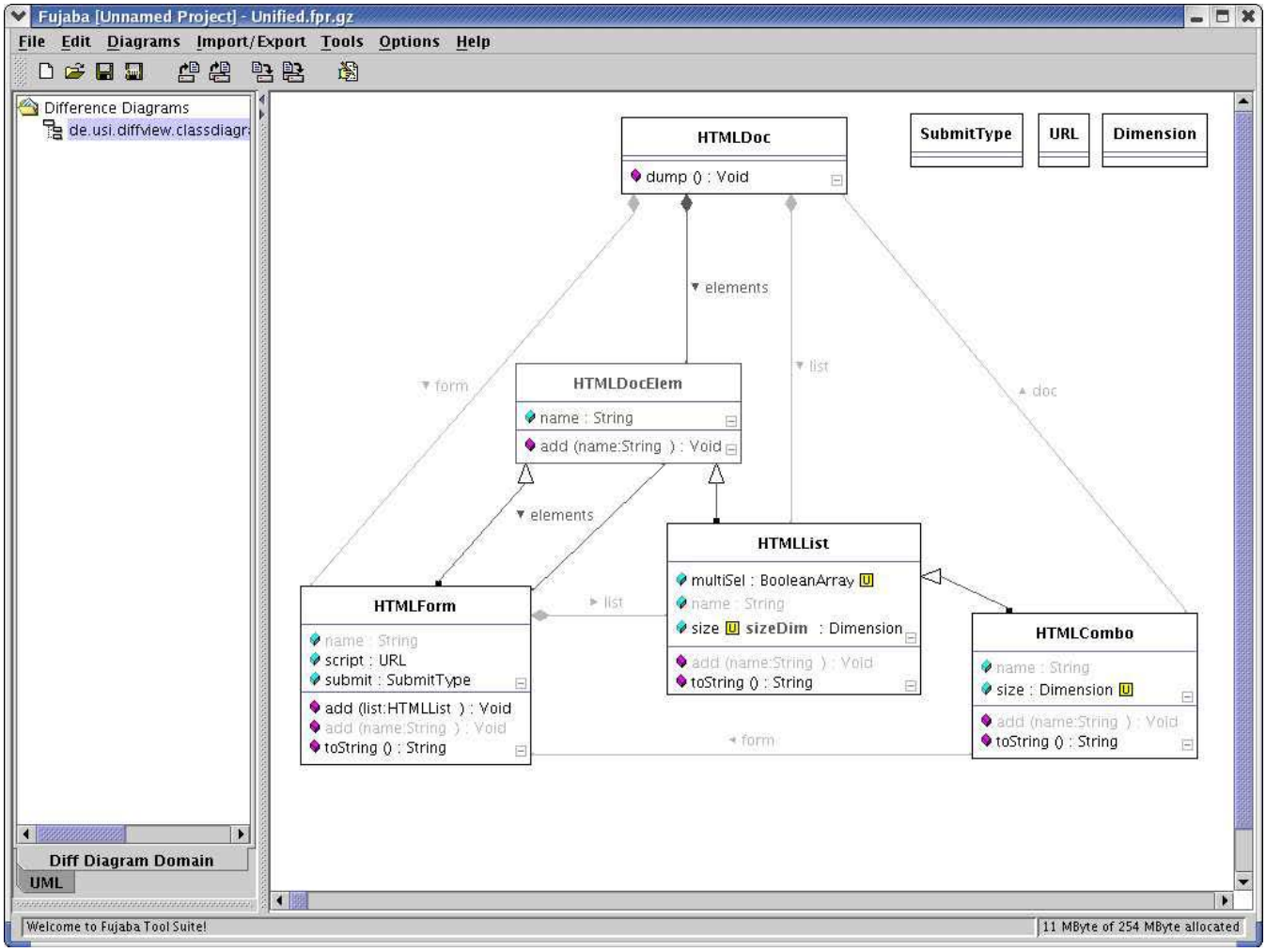

Figure 2: Difference class diagram.

and the method add(name:String) of class HTMLForm belong only to the left class diagram, whereas the attributes script and submit belong to both class diagrams and are shown in black color. A situation where a green or red class contains attributes and/or methods of a different color than the class itself can not occur, because a movement of attributes and methods from one class to another not similar class will not be identified by the algorithm. Such a movement is only detectable with a three-way-difference approach and not by a two-way-difference approach, see [6].

The third kind of differences are so-called *update differences*. Update differences are differences within one diagram element, i.e., attribute value changes of a diagram element. For example, the difference calculation algorithm has identified that two attributes of a class are similar, but the names are not identical. In addition, not identical visibilities, types and initial values of attributes result in update differences as long as the difference calculation algorithm identifies two attributes as similar. Otherwise it is a structural difference.

We visualize update differences by showing a small yellow button (u-button), cf. Figure 2. The u-button has three states. One state in which the value from the left class diagram is shown in red color, one state in which the value from the right class diagram is shown in green color and one state in which both values are shown side-by-side in red and green color, respectively. For example in Figure 2 the type BooleanArray of attribute multiSel in class HTMLList is the value in the left class diagram. In case of the attribute with type Dimension both name values are shown, i.e. size and sizeDim.[2]

3. TECHNICAL REALIZATION

The difference calculation algorithm and the visualization come along as Fujaba plug-ins. The difference calculation algorithm plug-in is coupled in such a way, that the algorithm is started by a menu entry in the tools menu. After selecting two XMI documents to be compared, the algorithm produces an output XMI document including the difference information. The last part of this process is to notify the visualization if required and pass the produced XMI document as parameter. This design allows us to ship also an

[2]The font looks a little bit bold.

independent tool, which can be used outside of Fujaba, e.g. as extension tool to commercial UML tools.

The second plug-in is the visualization plug-in. As we have started with the project we wanted to use the meta-model enhancement facilities [1] to reuse the already existing diagram meta-models for class diagrams, statecharts, activity diagrams, etc. Unfortunately, Fujaba's graphical user interface library, i.e. the FSA library, can not deal with different visualizations in different contexts. The idea was to display classes in a conventional class diagram context as they are and in a difference view context as classes with difference information such as red and green color. Enhancing the FSA library would have been resulting in a complete redesign of the library and an adaptation of all existing diagrams and plug-ins using the library. Therefore we decided to cut-out the meta-model for each diagram and copy it to our plug-in. In addition, we had to relax some restrictions concerning association types and cardinalities, e.g. difference class diagrams may contain classes with the same name and the same package, which was not allowed in the original meta-model.

Today, we support difference class diagrams and therefore the visualization plug-in consists of copies of nearly all meta-model elements representing class diagrams such as DiffUMLClass, DiffUMLAttr, DiffUMLAssoc, etc. We also developed a generic solution to add unparsing facilities for the difference information, which enables us to easily modify existing meta-models to display difference information.

In order to test our calculation algorithm, we developed a plug-in, that exports the pure model information of Fujaba class diagrams as XMI document. In general, we export all visible class diagram elements and leave out all other elements, e.g. get- and set-methods. The exported model is consistent, which means that we also export all necessary data types as well as all stereotypes. The XMI document format is compatible with Poseidon's XMI format [3]. Therefore our difference calculation algorithm is able to work with XMI documents produced either by Fujaba as well as by Poseidon.

4. FUTURE WORK

The short-term target is to complete the visualization plug-in and to enhance the plug-in with additional kinds of diagrams. We plan a sequence such as statecharts, activity diagrams and collaboration diagrams. An integration of additional diagram kinds will be long-term targets. Equipped with our generic difference unparsing facilities such an integration should be easy. In particular, we will test our developed facilities by integrating statecharts, first. In addition, we hope to benefit from the UML 2.0 MOF based meta-model and XMI export, which will produce a normalized XMI document and not specific documents from Rose, Fujaba or Poseidon.

Our long-term target is to develop an interactive merging process. Hence our difference calculation algorithm uses heuristics, which have been optimized for a certain set of diagrams but will also produce bad results on other ones. The idea is to integrate the developer in the process and benefit from his/her changes. Therefore we also support the developer with specific editing operations either for setting similarities as well as for merging elements. A first step is the n-button and u-button, which allow the developer to select one of the two available alternatives from the different diagrams. The result of the process will be a consistent merged class diagram.

5. ACKNOWLEDGMENTS

Special thanks to Jürgen Wehren, who is developing the difference calculation algorithm and to Stephan Lück, who is integrating the difference view plug-in into the Fujaba Tool Suite.

6. REFERENCES

[1] S. Burmester, H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool integration at the meta-model level within the fujaba tool suite. In *Proc. of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, (ESEC / FSE 2003 Workshop 3)*, pages 51–56, September 2003.

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

[3] Gentleware. *Poseidon for UML. Online at http://www.gentleware.com (last visited June 2004).*

[4] IBM. *Rose, the Rational Rose case tool. Online at http://www.rational.com (last visited June 2004).*

[5] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[6] D. Ohst. *Versionierungskonzepte mit Unterstützung für Differenz- und Mischwerkzeuge*. PhD thesis, University of Siegen, Siegen, Germany, 2004. in german (to appear).

[7] D. Ohst, M. Welle, and U. Kelter. Difference tools for analysis and design documents. In *Proc. of the IEEE International Conference on Software Maintenance 2003 (ICSM2003), Amsterdam, The Netherlands*, pages 13–22. IEEE Computer Society Press, 2003.

[8] C. Schneider, A. Zündorf, and J. Niere. Coobra - a small step for development tools to collaborative environments. In *Proc. of the Workshop on Directions in Software Engineering Environments (WoDiSEE), Edinburgh, Scotland, UK*, May 2004.

[9] P. Selonen. Set operations for unified modeling language. In *Proceedings of the Eight Symposium on Programming Languages and Tools, SPLST'2003, Kuopio, Finland, June*, pages 70–81. Kuopio, Finland: University of Kuopio, 2003.

An Adaptable TGG Interpreter for In-Memory Model Transformations *

Ekkart Kindler, Vladimir Rubin, Robert Wagner
Software Engineering Group, Department of Computer Science, University of Paderborn
[kindler|vroubine|wagner]@upb.de

Abstract

Triple graph grammars are a technique for defining correspondences between different kinds of models. Triple graph grammars can even be used for implementing translations and maintaining consistency between two models. But these implementations work only for automatically generated models. Therefore, the transformations cannot be applied to third-party models. In this paper, we discuss this problem and ideas for its solution.

1. Introduction

With the advent of *Model Driven Architecture* (MDA) [5], generation and transformation of models have become more and more important. There are many different techniques for defining and implementing such transformation; for a good overview and a discussion of the different approaches see [2].

Triple graph grammars (TGGs) [9], an extension of graph rewriting [7], is one of these techniques. TGGs are particularly useful for graph based models such as diagrams. A TGG defines a translation on a relatively high level of abstraction based on the syntactic structure of the underlying models. This way, it is possible to prove the correctness of the defined translation. In addition, TGGs do not only define a translation from one model to another, but also capture the correspondence between the source and the target model. Therefore, they can be used to translate back to the source model after some changes of the target model, and they can be used to check and maintain the consistency between two models.

TGGs have been used in different projects, and there are different implementations of TGGs. To apply TGG rules in PROGRES [8], they are translated to simple graph rewriting rules, which are then applied to the complete model stored in a database. FUJABA [10] applies Story Charts and pattern matching for this transformation. This results in a simple and quite natural implementation. This implementation, however, requires that there is a model on top of which the TGG rules are formulated; in fact there are three models, one for the source, one for the target and one for the correspodence part. Moreover, it is necessary that the models for which the rules are to be applied are generated from this meta model according to the rules of FUJABA. When dealing with models of third parties, where the mapping from the meta model to a Java implementation differs from FUJABA's implementation, this implementation does not work anymore.

In this paper, we present an idea for applying TGG transformations to models that have not been generated from the meta model underlying the TGG rules. Rather, we would like to use any (Java) implementation of the model. We call these models *in-memory models*. In order to apply a TGG translation to such in-memory models, there must be a mapping which defines how the constructs of the meta model, its classes, its attributes, and its associations are implemented. Though it would be a worthwhile task to develop a framework for defining such mappings in the most general way, we propose a simple technique to start with, which can be extended in the future. The idea is quite simple: The mapping is implemented by a class with a particular interface. This interface requires methods, which map arbitrary objects from the implementation to the corresponding class of the meta model. And it requires methods that, for a given object, provides all links corresponding to an association of the meta model. Moreover, this class must provide methods for generating objects and links in the implementation. With this additional *mapper class* it is easy to translate models by a TGG interpreter. In fact, there are two mapper classes: one for the source and one for the target meta model.

The ideas of this paper were inspired by a project and tool called *Component Tools*. For understanding this background, we will briefly discuss this project in Sect. 2. Then, we will rephrase the concept of TGGs in Sect. 3. The core ideas and implementation techniques for an in-memory TGG transformation will be discussed in Sect. 4.

2. Tool Support for System Engineering

In this section we give a brief overview on the concepts for a tool called COMPONENTTOOLS (see [3] for a more detailed description). Parts of this tool have been implemented as a prototype already. The tool will support building a system from components, transforming these models, and for exporting them for analysis purposes as well as for importing analysis results back to the component view.

COMPONENTTOOLS was originally inspired by the case study within the ISILEIT[1] project. The ISILEIT project aims at the development of a seamless methodology for the integrated design, analysis, and validation of distributed production control systems. Its particular emphasis lies on reusing

*This work has partly been supported by the German Research Foundation (DFG) grant GA 456/7 ISILEIT as part of the SPP 1064.

[1] ISILEIT is the German acronym for "Integrative Specification of Distributed Production Control Systems for the Flexible Automated Manufacturing".

existing techniques, which are used by engineers in industry, and on improving them with respect to formal analysis, simulation, and automatic code generation.

The specification of such systems is done in close cooperation with mechanical and electrical engineers. It turned out that system engineers prefer to construct systems from some components in a way that is independent from a particular modelling technique. However, they still would like to use the power of different techniques, once they have constructed their system.

Faced with this requirements, we have started to build a tool solving these problems. In the following, we will use a simplified toy train example representing a material flow system within our case study for explaining the main ideas.

Figure 1 shows such a simple toy train system built from components. There are basically four different components: straight tracks, curved tracks, tracks with a stop signal and switches. The components are equipped with some ports, which are graphically represented as small boxes or circles at the border of the component. The ports are used to connect the components to each other. In our example, there are ports representing the physical connections of tracks, and there are ports which allow to attach controller components, e.g. for the switches or the light signals. Note that, for simplicity, in our example we did not connect the system to controller components and only the physical connections of tracks are presented.

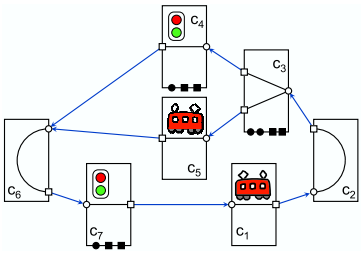

Figure 1: A toy train build from components

In order to build such a system, we need to provide a component library, which contains these four components. The component library defines all the available ports, their graphical appearance, and how ports may be connected. Moreover, the component allows us to provide a model for each component that defines its dynamic behaviour.

Figure 2 shows the Petri net models for two of the components. In fact, we can provide even more models for each component. For example, there could be abstract models as shown in Figure 2, or there could me more concrete ones. Or there could be additional models in different notation such as State Charts or other notations. From these models, and the system built by the user, ComponentTools generates several overall models of the system, each in one particular notation, which can then be used by appropriate tools supporting this formalism, e.g. for analysis, verification or code generation.

The presented tool is implemented in some parts as a prototype [4], whereas the model generation and transformations will be implemented in a course called "project group" at the University of Paderborn.

In the next section, we give a short introduction to triple graph grammars, which we are using for the specification

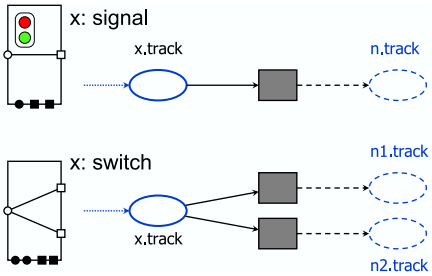

Figure 2: Two models for components

and execution of model transformations between the component model and the underlying models of each component to the overall models in a particular notation.

3. Triple Graph Grammars

In his original work, Schürr [9] extended pair grammars [6] to triple graph grammars. In contrast to pair graph grammars, triple graph grammars support context-sensitive productions with rather complex left-hand and right-hand sides. Generally, the separation of correspondence objects enables the modeling of m-to-n relationships between related sides.

The triple graph grammar approach makes a clear distinction between source and target models; it also keeps the extra links needed for specifying the transformations as a separate specification.

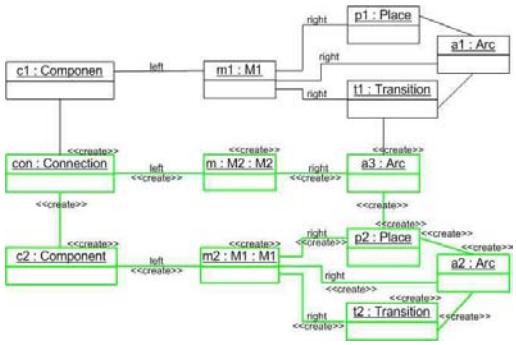

Figure 3: TGG Rule Example

A triple graph grammar specification is a declarative definition of a mapping between two meta models. In Fig. 3 a triple graph grammar rule is shown, defining the correspondence between a component and a Petri net model. It consists of a triple of productions (left production, correspondence production, right production), where each production is regarded as a context-sensitive graph grammar rule. The left production shows the generation of a new component and linking it to the existing one. The right part shows the addition of a new place, a transition and two arcs to the existing Petri Net. The correspondence production shows the relations between the left-hand and right-hand sides.

This declarative specification can be translated into simple graph rewriting rules which are used for the transformation in both directions. In Fig. 4 the forward transformation rule is presented.

The forward transformation rule is applied to the model, if the left production of the triple graph grammar is detected,

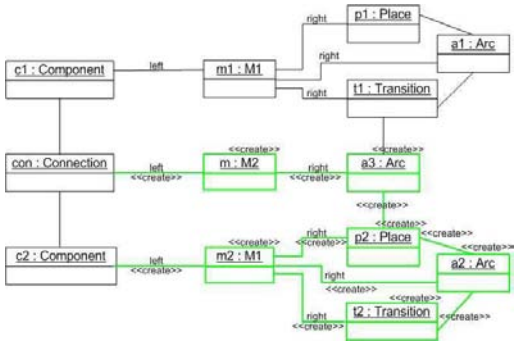

Figure 4: Forward graph rewriting rule

i.e. if a component was added to the project. In this case, the graph rewriting system will search for all objects contained in the left-hand side of the forward transformation rule. If a match is found, the correspondence objects, the objects representing new places, transitions, arcs, and links between the objects are created.

In contrast to the forward rule, the rule which handles the translation from the right-hand model to the left-hand model, i.e. from the Petri net model to the component model, is called backward rule. It is created from the triple graph grammar in the very same way as the forward rule: we just exchange the left and right side. For more details, we refer to [9].

The advantage of triple graph grammars over the other approaches lies within the definition of inter-graph relationships, which provide the flexibility to use productions for both forward and backward transformation and correspondence analysis. A triple graph grammar, as a declarative definition of the mapping between the two graphs, can be used for the implementation of a translator in either direction. Such an translator will be presented in the next section.

4. Interpreter

In this section, we present the core ideas and implementation techniques for an in-memory model transformation based on triple graph grammars. Before explaining our approach, we give a brief description of FUJABA's approach for graph rewriting and discuss its limitations.

The problem. In order to execute triple graph grammar rules in FUJABA, the specified rules are transformed into simple graph rewriting rules. These graph rewriting rules are translated to a Java implementation which performs the desired graph pattern matching and graph rewriting. However, this implementation is based on the meta models of the source, the target, and the correspondence graphs and requires that both meta models are implemented in a predefined way.

In fact, FUJABA requires the implementation to be automatically generated from the meta models by FUJABA. For example, each attribute of a meta model class must be implemented as a private variable with appropriate get and set methods. Associations must be implemented as bidirectional references with well-defined access methods following some naming conventions. These access methods allow navigation between in-memory objects, accessing and modifying in-memory objects, and creating new objects. The mapping between the conceptual model and its implementation is implicitly given by the code generator, which is fundamental for FUJABA's graph rewriting algorithm.

In the COMPONENTTOOLS project, we deal with already existing third-party models. In some cases, the source code of the model implementation is given. In other cases only some kind of an Application Programming Interface (API), which typically differ from FUJABA's model implementation. Hence, the graph pattern matching and graph rewriting algorithms of FUJABA will not work for these models.

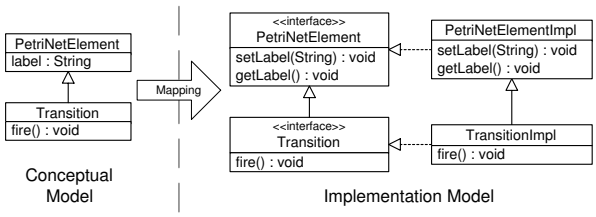

Figure 5: Simple mapping example

Figure 5 shows a part of an example of a *conceptual model* underlying the rules of a TGG and a typical *implementation*, which, for simplicity, is represented also in UML. This example shows that we cannot be even sure that the names of elements in the conceptual model have the same names in the implementation. In order to generate new objects, the TGG interpreter needs to know the names of the classes implementing the interfaces and how associations are implemented.

Architecture. In general, there is no way to map some objects and references of an implementation to the corresponding classes and associations of the conceptual model fully automatically without providing additional information. Therefore, we need a mechanism that defines this mapping such that the TGG interpreter can understand the implementation model. To this end, we propose a simple architecture which is shown in Fig. 6.

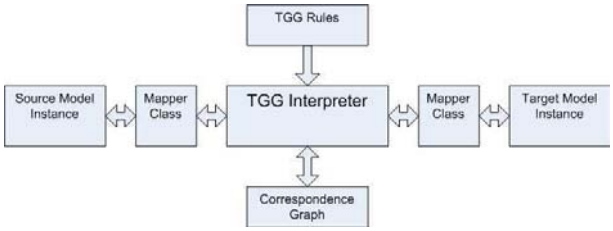

Figure 6: Architecture with Mapper Class

Between the TGG interpreter and the source and target model, there are *mapper classes*, which define the mapping between the conceptual model and the implementation. These classes provide methods that, for a given object of the implementation, return the corresponding class in the conceptual model. Moreover, they provide methods that return all links in the conceptual model for a given object of the implementation. On the other hand, the mapper classes provide methods for generating new objects in the implementation model for a given class of the conceptual model and they provide methods for generating links in the implementation.

For each conceptual model underlying the TGG and each implementation of such a model, a user must implement

such a mapper class. These classes are passed to the TGG interpreter as additional parameters. In order to pass these mapper classes to the TGG interpreter, the TGG interpreter defines a *mapper interface*, which must be implemented by all mapper classes. The interface requires that there are all the methods which have been discussed above: methods for mapping objects of the implementation to the corresponding classes of the conceptual model, methods for getting all links corresponding to some particular association of the conceptual model of an object of the implementation, and methods for generating objects and links in the implementation models that correspond to some class or association of the meta model.

Discussion. Though the mapper class approach is quite simple, it is the most powerful one because, in principle, any mapping can be implemented as a mapper class. The disadvantage of this approach is that it requires programming the mapper classes for each new implementation of a model, which is tedious work. In particular, an inexperienced user might provide a flawed mapper class, which would result in flawed translations even if the TGG interpreter works correctly.

Therefore, it would be nice to define the mappings from the conceptual model to the implementation on a higher level of abstraction and in a notation particularly tailored for this purpose. A good notation for defining such mappings, however, needs more detailed investigation. Once such a notation is available, it is easy to implement a standard mapper class, which receives such a mapping definition as input and which uses the Java Reflection API for implementing the methods required by the mapper interface. With this generic mapper class, it will no longer be necessary for the user to implement a mapper class for each new implementation. Rather, it will be necessary to provide an abstract definition of the mapping in the new notation.

Likewise, the TGG interpreter could be easily used with implementations that are generated automatically from the conceptual models. In this case, the mapper classes could be generated automatically too. Then, it will not be necessary to implement mapper classes for generated models. For example, we could use JMI generated and reflective interfaces.

Another idea for implementing mapper classes would be to have a standard mapper class which is provided with some scripts for implementing the mapping. Then, it would not be necessary to implement a complete mapper class; rather it is necessary to provide some scripts for defining the mapping only. For example, we could use the scripting language BeanShell [1] for this purpose. On the one hand, this approach would avoid the compilation step for the mapper class, which might be an advantage for a stand-alone tool. On the other hand, using a scripting language will result in some performance loss in comparison to a programming language.

Anyway, all these extended mapping concepts can be built on top of our mapper class concept by implementing a generic mapper class.

Implementation. Currently, we are working on an implementation of the above ideas in the context of ComponentTools. But, the TGG interpreter itself will be completely independent from the graphical user interface, so that it can be easily used in other tools such as Fujaba or as a stand-alone tool.

Even more, our interpreter and mapping concept can also be used for graph rewriting because triple graph grammars are just a specialized sort of graph grammars. The mapping concept immediately carries over to graph rewriting.

5. Conclusion and Future Work

In this paper, we have presented the problem of applying TGG transformations and consistency algorithms to in-memory models that have not been automatically generated. We have presented some ideas for an interpreter for TGGs that solves this problem. This way, TGG techniques can be applied to legacy code and models that have not been generated from our own models.

We just started with a detailed design of the mapper interface and with an implementation of the in-memory TGG interpreter. But, we hope to have a first prototype soon.

Acknowledgments

We would like to thank all members of the project group Component Tools at Paderborn University for all their discussions, which help to clearly identify the problem and to come up with the first concepts of the in-memory TGG interpreter.

References

[1] BeanShell. *Leightweight Scripting for Java.* http://www.beanshell.org (last visited July 2003).

[2] K. Charnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA 2003 Workshop on Generative Techniques in the Context of Model-Driven Architecture, Anaheim, CA, USA*, October 2003.

[3] A. Gepting, J. Greenyer, E. Kindler, A. Maas, S. Munkelt, C. Pales, T. Pivl, O. Rohe, V. Rubin, M. Sanders, A. Scholand, C. Wagner, and R. Wagner. Component Tools: A vision for a tool. In preparation, July 2004.

[4] J. Greenyer. Maintaining and using component libraries for the design of material flow systems: Concept and prototypical implementation, October 2003.

[5] OMG. *Model Driven Architecture.* http://www.omg.org/mda/.

[6] T. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences 5*, pages 560–595, 1971.

[7] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, Singapore, 1999.

[8] A. Schürr. PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems. Technical Report AIB 94-11, RWTH Aachen, Germany, 1994.

[9] A. Schürr. Specification of graph translators with triple graph grammars. In *Proceedings of the* 20^{th} *International Workshop on Graph-Theoretic Concepts in Computer Science*, Herrschin, Germany, June 1994. Spinger Verlag.

[10] University of Paderborn, Germany. *Fujaba Tool Suite. Online at http://www.fujaba.de/.*

Standardizing SDM for Model Transformations

Hans Schippers
Aspirant FWO - Vlaanderen
Formal Techniques in Software Engineering
Universiteit Antwerpen, Belgium
Hans.Schippers@ua.ac.be

Pieter Van Gorp
Formal Techniques in Software Engineering
Universiteit Antwerpen, Belgium
Pieter.VanGorp@ua.ac.be

ABSTRACT

Transformations are a key technology in model driven software engineering since they are used to implement refinements for platform independence, restructurings for software migration and weavings for aspect composition. By considering transformations as models, one can develop transformations in the same paradigm as conventional applications. In this paper, we illustrate how Fujaba's language for graph rewriting has been applied for the CASE tool independent development of model transformations.[1]

1. MODEL TRANSFORMATION IN FUJABA

As Sendall and Kozaczynski state [2], model transformation can be seen as the *heart and soul* of model driven software development. Model transformations therefore deserve to be treated as first class entities in software development.

Considering transformations as models [3], recent experiments [4] have shown that Story Driven Modeling (SDM [5]) can be used as a language for the visual development of refactorings (which are a particular kind of "horizontal" model transformations). However, SDM's implementation in Fujaba suffers from two significant problems. First, the SDM metamodel in Fujaba is non-standard and it is only implicitly present in the source code. As a consequence, only the Fujaba editor is suitable to create and store SDM instances. Second, the Fujaba code generator exclusively generates code conforming to non-standard conventions, meaning it can solely be deployed on the Fujaba repository.

2. A NEW SDM COMPILER BASED ON MDA STANDARDS

Both these problems can be overcome by making use of a few MDA standards. More specifically UML, as an alternative for SDM, MOF for standardized (meta)model access and storage, and finally JMI as the binding between MOF and the Java programming language.

[1]A more elaborated paper on this work has been accepted at the ICGT'04 workshop on Software Evolution through Transformations [1].

2.1 UML Profile for Model Transformation

The first issue has been tackled by designing a UML profile for SDM, the aim being to resemble the SDM concrete syntax as closely as possible, while keeping the semantics in place. Thus, an attempt was made to associate each SDM construct with a suitable UML counterpart. To handle different variations of the same construct (such as *for each* activities versus *code* activities versus normal story activities), UML stereotypes have been used to make the distinction.

Mapping the *control flow* part of SDM proved to be fairly straightforward, because of the presence of activity diagrams in the UML standard. For the Story *primitives*, the closest match were object (collaboration) diagrams. However, these don't seem to be available in every CASE tool, and even if they are, they often offer less visual features (such as displaying attribute assignments), than class diagrams. Therefore, the latter were the preferred candidate.

The fact that the semantics of UML class diagrams in the context of model transformation differ somewhat from their conventional usage, does not really pose a problem, as people probably know what context they are dealing with. Furthermore, the model transformation semantics are formalized in OCL, although this currently needs to be checked in a separate tool, as CASE tools typically do not allow the addition of meta-constraints yet. As an illustration, Table 1 lists part of the actual mapping of SDM to UML.

SDM Construct	UML Construct
Story Activity	ActionState
ForEach Activity	ActionState with «for each» stereotype
Unbound object	UmlClass
Bound Object	UmlClass with «bound» stereotype

Table 1: Extract from SDM to UML mapping

It should be clear that any UML compliant CASE tool can now be used to create SDM instances. Additionally, since UML complies to the MOF [6] standard, any MOF repository can be employed for storing the models in a standardized way. For example, the NetBeans MetaData Repository (MDR [7]) is an open source Java implementation of MOF (or JMI [8] to be more precise) that is used in several UML and MDA tools [9, 10]. Note that the latter does not put any additional requirements on the CASE tool being used. Having a CASE tool which makes use of a MOF repository internally, is certainly convenient if it makes its API accessible to the code generator, but this can easily be circumvented by exporting UML models to XMI, and importing the result in an external MOF

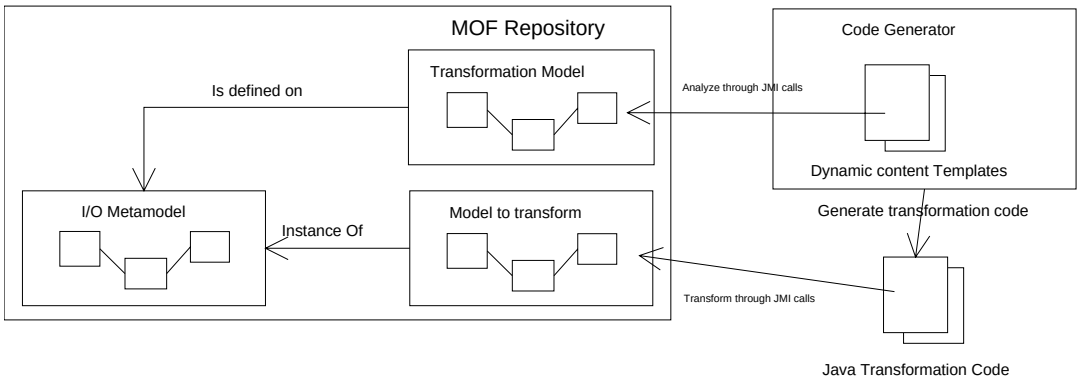

Figure 1: Code Generation Process

repository. Obviously, in that case, XMI export capabilities need to be available in the CASE tool in question, but the majority of tools do provide this feature nowadays.

2.2 Generation of Transformation Code

To solve the second problem, the Fujaba code generator was replaced by another open source solution called AndroMDA [11] for two reasons. On the one hand, AndroMDA was designed to get the information necessary to generate code from MOF compliant models inside a MOF repository. On the other hand, at the heart of the code generator is a set of dynamic content templates, which can easily be replaced in order to support different target platforms.

The code generation process is depicted in Figure 1. The MOF repository (MDR) could be seen as the starting point, as it holds the transformation specification (transformation model). The dynamic content templates contain directives in order to extract information from this model, and based hereon, deliver a java source file with the actual transformation code. A sample of such a template can be found in Figure 2. It handles a UML ActionState (the *state* variable), by first checking what kind of activity it actually is, and then performing the appropriate action. To find out the required information, the template calls upon a helper object, accessible via the *transform* variable, which does the actual querying of the transformation model. The result consists of all text not surrounded by any type of brackets (just *_jcmtg_success* in this case), as well as the (string-formatted) result of any calls surrounded by ${}.

```
1        <#if transform.isCodeState(state)>
2                <#-- Code state => process its entry action
3                        and put the result here -->
4                <#list transform.getProcessedStatements(state) as statmnt>
5        <@indent/>              ${statmnt}
6                </#list>
7        <#elseif transform.isLinkState(state)/>
8        <@indent/>              _jcmtg_success${nd} =
9        <@indent/>                      ${transform.getMethodCallEntryAction(state)};
10       <#else/>
11               <#-- State with Transformation Primitive (diagram) -->
12               <#local primPkg = transform.getTransPrimitivePackage(state)/>
13               <#include "TransPrimitive.ftl">
14       </#if>
```

Figure 2: Dynamic Content Template Sample

At the moment, only one set of templates is available, which is responsible for the generation of Java code conforming to the JMI standard, which is a mapping of MOF to Java. In other words, the resulting code is capable of accessing a JMI-compliant MOF repository such as MDR, and performing the actual transformation upon a model stored within there. Thus, model transformation code can be generated for any model, instantiating any metamodel, stored in a MOF repository which can be accessed through JMI interfaces, while other configurations (like EMF [12]) can be supported by writing a new set of templates. Note that currently, only intra-metamodel transformations are supported, that is, transforming an instance of a certain metamodel into another instance of the same metamodel.

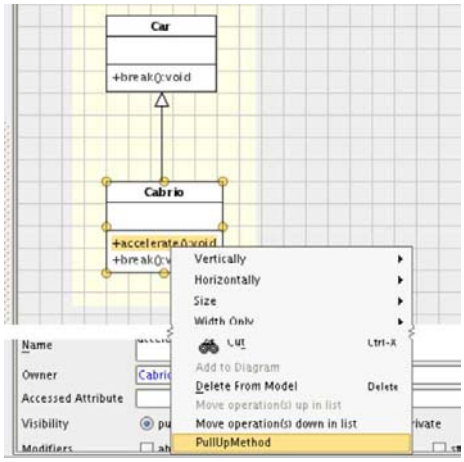

Figure 3: Refactoring Plugin generated for Poseidon.

3. EXAMPLE TRANSFORMATION

In order to validate the CASE tool independent approach to transformation development, we used the MagicDraw UML tool [13] to specify the “Pull Up Method” refactoring that illustrated the original Fujaba approach in [4]. Figure 3 shows the plugin that the new compiler generated for the Poseidon UML tool [9]. This comprises the model transformation code on the one hand, and some plumbing code on the other hand, the latter generated by a separate so-called AndroMDA “cartridge”. Obviously, Poseidon could be used for the specification as well, instead of MagicDraw, but we wanted to stress interoperability between several tools.

Figure 4 visualizes that the actual transformation should only be applied if the input model satisfies a certain precondition (The «link» stereotype merely indicates that the specification of this precondition is to be found in a separate diagram, pointed to by means of a

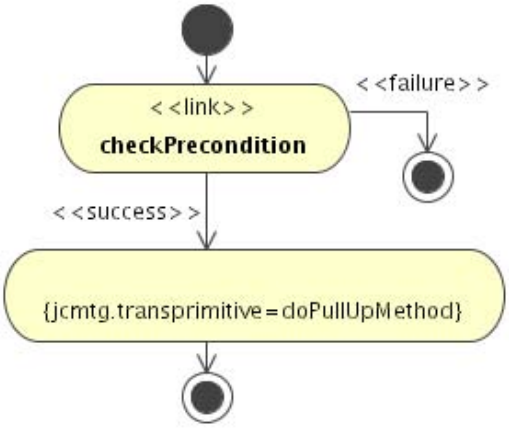

Figure 4: Transformation Flow edited in MagicDraw.

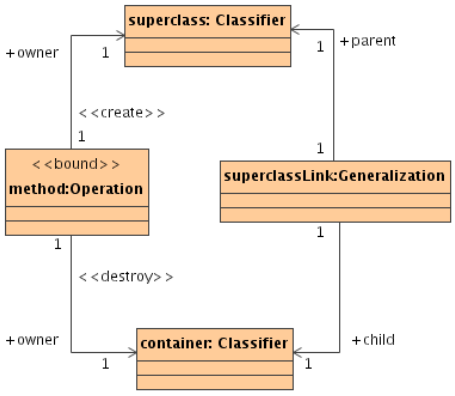

Figure 5: Transformation Primitive edited in MagicDraw.

tagged value). Basically, the precondition is satisfied if and only if it makes sense to perform a Pull Up Method. In other words if, for example, the class owning the method in question, does not have a superclass, the precondition would fail. Equivalent to Fujaba, the underlying semantics are based on programmed (or controlled) graph rewriting [14].

Figure 5 displays the primitive graph transformation rule that removes a method from its containing class and adds it to the list of methods from the superclass: The bound object named *method* represents the method which should be pulled up, and serves as a starting point for the lookup of its containing class *container* via the UML meta-association *owner*. Once *container* is bound too, its superclass is looked up in a similar way. The precondition guarantees that it is indeed possible to bind all objects. At that point, the owner of the method is changed from *container* to *superclass*, indicated by the «create» and «destroy» stereotypes. This completes the transformation.

Some problems, for example complex numerical calculations, seem to be solved more easily when using a conventional programming language like Java than when using graph rewriting. To illustrate that one does not have to choose for one approach exclusively, a part of the precondition is currently implemented using Java that integrates with the generated part of the transformation. It can also be specified completely using visual SDM constructs equivalent to those in Figure 4 and 5.

4. CONCLUSIONS AND FUTURE WORK

We can conclude that the model driven engineering techniques for platform independence can be applied to model transformations as well. This enables the developers of refactorings, normalizations, refinements and other kinds of program transformations to bypass CASE tool vendor lock-in.

Future work that will affect MDA and graph rewriting practitioners includes the integration of OCL into SDM, the possible support for additional visual language constructs, repository-platform independent Java transformations and the development of a dedicated GUI. Concerning the latter, it would be interesting to replace Fujaba's code generators for class and story diagrams by the discussed open source alternatives since it will reuse Fujaba's powerful editor. At the same time, Fujaba's models would be more interchangeable by reusing mature XMI serializers from NetBeans' MDR. Finally, the generated code could be deployed to any JMI repository and support for other repositories could be added with moderate effort.

The OCL integration would improve the transformation tool in two ways. Firstly, it would remove the current dependency on an external OCL tool to evaluate the well-formedness of model transformations. Secondly, constraints within the transformation model could be expressed not only in plain Java but also in OCL. The hybrid graphical/textual language would support the complete specification of repository independent model transformations.

The code driven Java approach, described in the context of the precondition of our example in Section 3, currently suffers from direct dependence on the framework of the target repository. Building Java wrappers [15] around the metaclasses of different repository platforms (JMI, EMF, ...) is a solution that is not efficient in terms of developer effort when these classes are implemented manually. One could follow a hybrid model/code driven approach by generating such wrappers on the one hand and writing pieces of manual transformation code that use the generated wrappers on the other hand. The current compiler does not use such a wrapper-based approach but generates code that calls the repository-platform specific metaclasses directly. However, if the Java "backdoor" turns out to be desirable in the long term, one can extend the compiler to generate wrappers that could be used to write repository-platform independent transformation fragments in Java.

Future work that will affect maintainers of the SDM compiler includes applying vertical model transformation on the compiler itself. It is motivated by the objective to reuse the graph matching and rewriting algorithm for different target repositories and to make the dynamic content templates as trivial as possible. This work should lead to more insight into how one can maximally reuse and specialize parts of MDA code generators.

5. ACKNOWLEDGMENTS

This work has been sponsored by the Belgian national fund for scientific research (FWO) under grants "Foundations of Software Evolution" and "A Formal Foundation for Software Refactoring". Other sponsoring was provided by the European research training network "Syntactic and Semantic Integration of Visual Modeling Techniques (SegraVis)".

6. REFERENCES

[1] Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations, October 2004. Accepted at Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation.

[2] S. Sendall and W. Kozaczynski. Model Transformation - The Heart and Soul of Model-Driven Software Development. *IEEE Software, Special Issue on Model Driven Software Development*, pages 42–45, Sept/Oct 2003.

[3] Jean Bézivin and Sébastien Gérard. A preliminary identification of MDA components. In *Proc. Generative Techniques in the context of Model Driven Architecture*, 2002.

[4] Pieter Van Gorp, Niels Van Eetvelde, and Dirk Janssens. Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Metamodel. In *Proceedings of the 1st International Fujaba Days*, University of Kassel, Germany, October 2003.

[5] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, volume 1764 of *LNCS*, pages 296–309. Springer Verlag, November 1998.

[6] Object Management Group. Meta-Object Facility Specification, April 2002. version 1.4. document ID formal/02-04-03.

[7] Sun Microsystems. NetBeans Metadata Repository, 2002. <http://mdr.netbeans.org/>.

[8] Sun Microsystems. Java Metadata Interface Specification, June 2002. document ID JSR-40.

[9] Gentleware. Poseidon for UML, version 2.2, 2004. <http://www.gentleware.com>.

[10] Compuware. OptimalJ. <http://www.compuware.com/products/optimalj/>, 2004.

[11] M. Bohlen. AndroMDA - from UML to Deployable Components, version 2.1.2, 2003. <http://andromda.sourceforge.net>.

[12] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. International Business Machines, January 2004.

[13] No Magic. Magicdraw. <http://www.magicdraw.com/>, 2004.

[14] Horst Bunke. Programmed graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 155–166. Springer-Verlag, 1979.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 4, pages 139–150. Professional Computing Series. Addison-Wesley, 1995.

A MOF 2.0 Editor as Plug-in for *FUJABA*

Carsten Amelunxen
Technische Universität Darmstadt
Institut für Datentechnik, FG Echtzeitsysteme
Merckstr. 25
Darmstadt, Germany
carsten.amelunxen@es.tu-darmstadt.de

ABSTRACT

In this paper we describe how we build a MOF 2.0 editor as a plug-in for Fujaba. The new versions of UML and MOF offer new concepts for structural modeling. We will use these new concepts to generate metamodels for several domains in compliance with common standards like MOF 2.0, JMI, OCL. We figure out how our efforts can be used as a starting point to improve Fujaba with regard to UML 2.0. Finally we present how we implemented the plug-in, what kind of technologies and components we used and what we will achieve with our efforts.

1. INTRODUCTION

The new upcoming version of the Unified Modeling Language (UML) [7] introduces new modeling concepts. Beside the new version of the UML there is also a new version of the Meta Object Facility (MOF) [1] which strongly depends on UML. The part of the UML specification that describes structural modeling (UML infrastructure [6]) is adopted by MOF. So MOF comprises the new UML constructs for structural modeling, too. We want to use MOF 2.0 and decided after a period of evaluating alternatives to realize our approaches in Fujaba.

One aim of our contributions for the development of Fujaba is to realize a MOF 2.0 plug-in which is able to generate metamodels for several domains. The generated metamodel implementations should comply with the Java Metadata Interface (JMI) [4] a standard provided by SUN. We want to use the generated metamodels in several fields like tool integration and re-engineering for example. Thus we have to be able to generate different kinds of metamodel implementations with a flexible code generation mechanism which uses MOF 2.0 compliant metamodels created with Fujaba as input.

The realization of our target plug-in or rather the actual stage of development as well as the benefits for Fujaba are subjects of this paper. The new features of MOF 2.0 compared to Fujaba are described in section 2 followed by an overview of further steps of development in section 3. Section 4 gives an overview on JMI. The application of JMI is described in section 5. Finally the conclusions and future work are part of section 6.

2. FEATURES OF MOF 2.0

The new features of the MOF 2.0 plug-in compared to the actual features of Fujaba's UML class editor mainly concern associations and packages. MOF 2.0 separates between real associations and implicit relations. Each class attribute may have an opposite attribute thereby realizing a simple bidirectional association. Such implicit relations are visualized by the MOF 2.0 plug-in as usual associations are visualized in the current Fujaba version. The separation of implicit relations and real associations gets clear regarding code generation. Implicitly related attributes would be mapped on code by using referencing attributes. A real association is mapped on an own class as demanded by the JMI specification. The separation has to be reflected in visualization and therefore real associations are depicted by a diamond node according to the MOF 2.0 specification (see figure 1). Each diamond node represents an association and each line connecting the association with the associated classes represents an association end. With such a mechanism we keep the opportunity to extend the plug-in for n-ary association although in MOF 2.0 only binary associations are allowed.

Other new features of MOF 2.0 compared to Fujaba are the relations between association ends. MOF 2.0 offers the opportunity to declare an association end as redefinition, subset, or union of other association ends (see figure 1). Those relations enhance the possibilities of modeling by specifying the relation between associations which are indirectly related by the type compatibility of their associated classes. Considering the example in figure 1 it is also possible to associate an instance of class HardwareDeveloper with an instance of class HardwareProject by using association Develop instead of association DevelopHardware due to the type compatibility of the associated classes. Such an implicit relation can now explicitly be marked as redefinition or subsetting. Without those constructs it would not be possible to express that a hardware developer develops only hardware (without using OCL constraints). The redefinition of association end developer by the association end hwDeveloper ensures that a hardware developer just develops hardware projects by suppressing the instantiation of the association develops between an instance of class Project and an instance of class HardwareDeveloper.

Another important feature is the possibility to define an association end as subset of another association end. The subsetting of an association end causes the propagation of link instances from the subsetting association end to the subsetted association end. In the example of figure 1 the query on association Develop for all projects for a specific hardware developer returns all projects that have been reg-

istered as instance of association DevelopHardware although there have never been an instantiation of association Develop with those instances. The linkage between both association is caused by the subsetting.

In addition to the mechanism of subsetting it is possible to declare a superset as exclusive union of its subsets. The definition of association end project as union of its subsets means that a developer can either develop hardware or software. The instances of the association ends hwProject and swProject are also part of the association end project. The union constraints prevents that there are instances of project beside the instances of the subsets. For further details see [6] or [2].

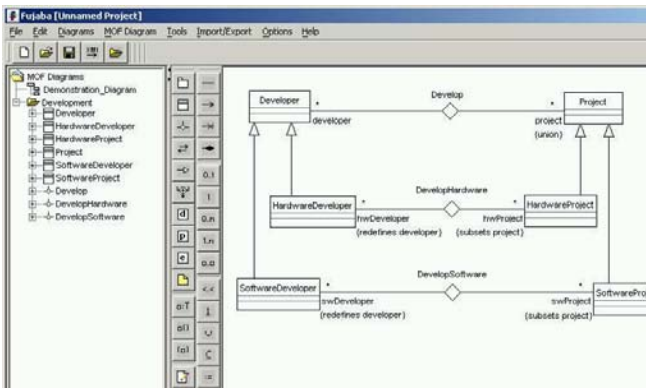

Figure 1: Screenshot of the MOF 2.0 plug-in

Furthermore MOF 2.0 offers the possibility to define dependencies between packages. Those dependencies (import, combine, merge) specify how packages are related to each other. Our work on the package editor has not proceeded far enough that an appropriate discussion is possible at the time of writing this paper. As a result of the packages and their significance in MOF 2.0 the current Fujaba package browser is not sufficient anymore. So we decided to build a new browser that supports package hierarchies and offers a higher degree of structuring.

Beside the new features which are undoubtedly an enrichment of available modeling facilities there are also useful features in the current version of Fujaba that are not available in MOF 2.0. For example, in MOF 2.0 class attributes are public or private. There is no possibility in MOF 2.0 to define an attribute as protected nor as static or final. That problem can be solved by the use of the adapter pattern as described in [9].

Table 2 summarizes some significant features and their availability in UML 1.x, Fujaba's UML, MOF 2.0 and UML 2.0. Obviously the most desirable modeling language concerning the widest range of modeling constructs is UML 2.0. There is a big gap between the possibilities UML 2.0 offers and the possibilities implemented by Fujaba as well as between Fujaba and MOF 2.0.

3. FURTHER STEPS OF DEVELOPMENT

Our work on MOF 2.0 can be regarded as an initial step for the evolution of Fujaba towards UML 2.0. Basically MOF

Constructs	UML 1.x	Fujaba	MOF 2.0	UML 2.0
private public	+	+	+	+
protected static	+	+		+
packages	+		+	+
package merge			+	+
kinds of package merges			+	
qualified associations	+	+		+
aggregations	+	+		+
composition	+	+	+	+
opposite - association			+	+
n-ary associations	+			+
related association ends			+	+
inheritance between associations	(+)		+	+
inner classes	(+)			+
dependencies	+			+
interfaces		+		+
reflection			+	

Table 1: Availability of modeling constructs

2.0 is a subset of UML 2.0 which covers the structural modeling of UML 2.0 (UML 2.0 Infrastructure [6]) expanded by some specific metamodeling features like extensibility and reflection. MOF 2.0 is a metamodeling language that can be used to specify modeling languages. The best known instances of MOF 2.0 are UML 2.0 as well as MOF itself. The features of MOF are limited compared to UML 2.0 but optimized with respect to the demands of metamodeling. So MOF represents a starting point in the definition of UML 2.0 and therefore can also be a good starting point for implementing UML 2.0. The mapping from MOF to Java (JMI) is designed regarding the demands of metamodels.

Thus a MOF 2.0 compliant Meta-CASE tool with a JMI compliant code generator is an appropriate basis to realize an extensible UML 2.0 CASE tool. Regarding those relations as described before we propose a scenario of evolution as follows.

1. First we realized an editor that is capable of generating code from an intersection of the modeling constructs of MOF 2.0 and the current Fujaba version. The graph transformations can be applied to the resulting modeling constructs. In the future we are interested to optimize the generated code with respect to the demands of embedded systems.

2. The second step will be the adaptation of the graph transformation for the generation of JMI compliant code. That code will be used for Meta-CASE applications.

3. The next major step is to expand the set of modeling constructs from an intersection of MOF 2.0 and Fujaba to a union of the concepts of both languages. There should also be both kinds of code generation

(JMI compliant code for Meta-CASE applications and target code for embedded systems) with regard to the different kinds of application.

Those steps are planned for the near future and are intended to result in a Fujaba that is based on UML 2.0 Infrastructure. At least we propose to keep MOF and UML separately due to different code generations and different purposes. For both versions of the UML Infrastructure the graph transformation has to be adjusted to the new concepts.

4. METAMODELING WITH JMI

One of the central aspects of the implementation of our MOF plug-in is the compliance to the Java Metadata Interface. Therefore we give a short overview of JMI. JMI defines a structure for the creation, storage, access and discovery of metadata by specifying a Java language mapping for MOF 1.4. JMI provides a common Java programming model for handling of metadata. This is done by the description of a set of interfaces which represents the reflective parts of MOF as well as the structural characteristics of MOF instances. The interfaces that cover the structure of the metamodels are divided into four categories. An example for a concrete JMI mapping which covers instances of all four categories is depicted in figure 2. It shows the mapping of a package with a binary association between two classes on JMI. The four categories are:

Package Objects create and manage instances of all included metaclasses. Instances are class proxy objects, association objects and package objects for nested packages. The package MetaModel in figure 2 is mapped to the interface MetaModel and the appropriate implementation MetaModelImpl. The instance of the package implementation is the initial point for the instantiation of the metamodel. There are accessor methods for all class proxy objects as well as for all association objects.

Class Proxy Objects act as a factory and as a container for creating and storing instance objects. There is only one class proxy object for each metaclass. The class proxy object of the metaclass Operation in figure 2 is of the type OperationClass. The class proxies are managed by the superior package instance.

Instance Objects represent an instance of the appropriate metaclass. Each instance object is created and stored by a class proxy object.

Association Objects are just like class proxy objects container for the handling of association instances. There is only one association object of the same type at runtime. The association objects are created and stored just like the class proxies by the superior package. Association instances are stored by the use of an unspecified link object contrary to the handling of class instances. In figure 2 the association Has is mapped to the interface Has with the appropriate implementation HasImpl.

For example the instantiation of the class Operation assumes an instance of the package implementation (MetaModelImpl)

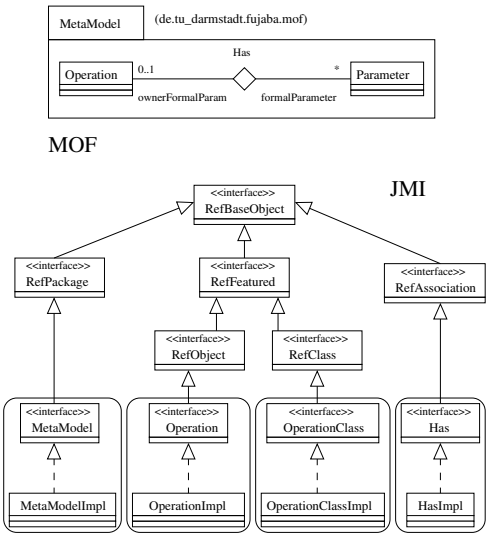

Figure 2: Example of JMI code mapping

which provides access to the class proxy objects. The class proxy object contains the factory method createOperation that returns an instance object. Finally, the instance object can be used to store instance scoped attributes. The instantiation of an association requires just like the instantiation of classes the request of an association object from the package object. Association instances (so called links) can be handled by using the association object's methods like add, remove etc.

A metamodel conforms to the JMI specification as long as the interfaces are satisfied. The implementations of the interfaces are just informally described and therefore may vary depending on different usages of the metamodel.

The currently existing JMI specification is designed to map MOF 1.4 compliant metamodels to Java. So the actual JMI specification does not cover the latest version of MOF. There are a lot of features in MOF 2.0 that force an extensive revision of JMI. For example, the current storage of association instances is not able to cover all features of MOF 2.0, actually it is not even able to cover MOF 1.4 properly. Such a revision is an essential motivation of our work on JMI.

5. A JMI-COMPATIBLE MOF 2.0 PLUG-IN

One of the central components of the MOF 2.0 plug-in is a code generator for JMI compliant metamodels. We need a code generator that is highly flexible and easily configurable for several target applications. Such a generator is the MOF Metamodeling Tool (MOmo) Compiler [3]. The MOmo Compiler is able to generate JMI compliant MOF metamodels from XMI [8] files. Its great advantage lies in its modularity concerning the architecture as well as the handling of several styles of target code by applying different sets of templates. Hence, the appropriate way to realize the Fujaba MOF plug-in was to bootstrap the metamodel by using MOmoC and a special rudimentary MOF 2.0 metamodel. The rudimentary MOF 2.0 metamodel for bootstrapping consists of just one package including the minimal set of necessary features. The distribution of the MOF 2.0

specification over several packages as done in the specification is planned for further iterations.

The first version of the plug-in's metamodel has been modelled in Rational Rose and passed to the MOmoC code generator via export of UML XMI. So the first metamodel that makes use of the new features of MOF 2.0 as done in the specification may not be introduced before the first bootstrap iteration due to the lack of MOF 2.0 features in common modeling tools. The way of realizing the plug-in is depicted in figure 3. Figure 3 shows a scenario with some of the main functions of the plug-in. The functionalities realized yet and needed for bootstrapping are marked in gray.

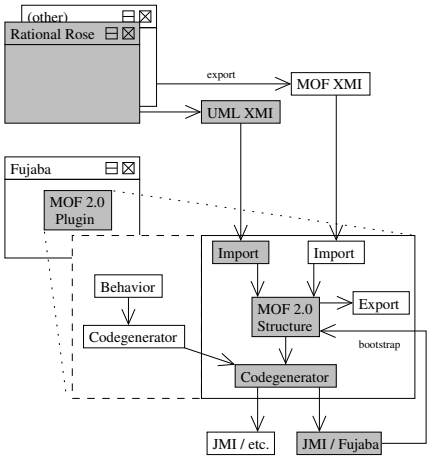

Figure 3: Scheme of the realized bootstrap scenario

The actual version of the plug-in at the time of writing this paper consists of a generated JMI compliant MOF 2.0 metamodel with a graphical editor for instantiating the most important elements except packages and their dependencies. The limitations are caused by the capabilities of the graphical editor which is based on the current editor for class diagrams in Fujaba. In general the metamodel is fully usable. The MOmo compiler has been modified to generate a metamodel with an implementation compatible with Fujaba. The code generator has been integrated into the plug-in and operates on the plug-in's metamodel. So the plug-in is able to generate its own metamodel. The further improvement of the metamodel will be done with the plug-in itself as soon as the graphical editor's degree of usability permits it.

The first versions of the generated metamodel consisted of an implementation that did not consider any cooperation with the graph transformation subsystem of Fujaba. It consisted just of an independent metamodel with all necessary actions and unparse modules that were essential as designated by Fujaba's plug-in mechanism. The metamodel implemented all interfaces demanded by JMI as described in section 4. But even such a rudimentary integration of a different metamodel caused a problem with the currently available plug-in mechanism. In the special case of a JMI compliant metamodel the instantiation is done by a factory method and not by a constructor as supposed by Fujaba's loading mechanism. So we had to modify Fujaba in a separate branch as described in [9] even in such an early state of integration.

Parts of our research activities also require the specification of behaviour in combination with the new features of MOF 2.0. Thus we had to enhance the metamodel's implementation with the intent to adopt Fujaba's graph transformation engine. The implementation has to apply a mechanism for cooperating with the graph transformation code generator. Such a mechanism is described by the adapter pattern [5]. The used adapter is depicted in figure 4. The former implementation OperationImpl of the JMI interface Operation as exemplarily used in figure 4 is replaced by a subclass OperationAdapter which implements all functions as demanded by JMI as well as all function demanded by Fujaba's graph transformation. There is an inheritance between Fujaba's graph transformation classes and the implementation of the metamodel. The instance of the related proxy class (OperationClassImpl) instantiates the adapter instead of the class OperationImpl. This is no violation of the JMI specification because the adapter still implements the JMI interfaces. For further details see [9].

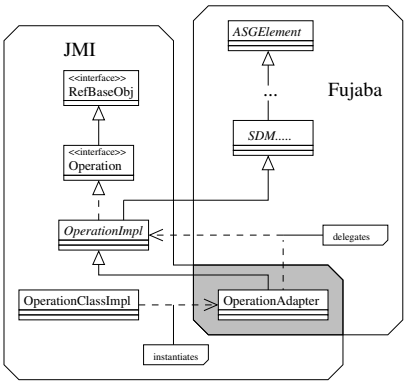

Figure 4: JMI Fujaba Adapter

6. CONCLUSIONS

The first version of our MOF 2.0 plug-in uses a rudimentary, but JMI compatible MOF 2.0 metamodel. Basically, the editor is able to generate its own metamodel. So further iterations of the metamodel will be done by using the plug-in itself. It is our demand to satisfy the MOF 2.0 specification including packages and their relations in detail although not in the first iteration. The primary concepts of MOF 2.0 have already been taken into account. All other features will follow in further iterations.

The plug-in is able to import UML XMI. There was no need for an import of MOF XMI yet. But the enhancement to MOF XMI is already planned for the future, when the plug-in will be able to write MOF XMI. It offers a code generation mechanism that is very easily expandable for the needs of other target frameworks just by maintaining several sets of templates. In this respect the integration of the Fujaba code generation is still an open issue.

Furthermore it is our intention to integrate an OCL compiler to enhance modeling capabilities as well as to improve the bootstrap process by considering the constraints of the MOF 2.0 specification. Additionally we will concentrate on the improvement of the generated metamodels.

Finally, there are three major advantages the Fujaba community will benefit from. First of all our new MOF plug-in offers the new features of MOF 2.0 and might act as a basis for upgrading Fujaba class diagrams to UML 2.0. One fundamental disadvantage of the current Fujaba version is the missing package concept which complicates the use of Fujaba in large projects. The package concept implemented by the MOF plug-in solves this problem and offers an easy way to organize large projects. Last but not least the compliance to standards like MOF and JMI opens up new application domains and therefore might expand the Fujaba community.

7. REFERENCES

[1] Adaptive Ltd, Ceira Technologies Inc., Compuware Corporation, Data Access Technologies Inc., DSTC, Gentleware, Hewlett-Packard, International Business Machines, IONA Technologies, MetaMatrix, Rational Software, Softeam, Sun Microssystems, Telelogic AB, Unisys, and WebGain. *Meta Object Facility (MOF) 2.0 Core Proposal*, April 2003. ad/2003-04-07.

[2] C. Amelunxen, L. Bichler, and A. Schürr. Codegenerierung für Assoziationen in MOF 2.0. In *Proceedings of the Modellierung 2004*, volume P-45 of *Lecture Notes in Informatics*, pages 149–168. Gesellschaft für Informatik, March 2004.

[3] L. Bichler. Tool Support for Generating Implementations of MOF-based Modeling Languages. In J. Gray, J.-P. Tolvanen, and M. Rossi, editors, *Proceedings of The Third OOPSLA Workshop on Domain-Specific Modeling*, Anaheim, USA, October 2003.

[4] R. Dirckze. *Java^{TM} Metadata Interface (JMI) Specification, Version 1.0*. Unisys, 1.0 edition, June 2002.

[5] Erich Gamma AND Richard Helm AND Ralph Johnson AND John Vlissides. *Entwurfsmuster*. Addison-Wesley, 1996.

[6] Object Management Group. *Unified Modeling Language: Infrastructure, Version 2.0*, September 2003. ptc/03-09-15.

[7] Object Management Group. *Unified Modeling Language: Superstructure, Version 2.0*, April 2003. ad/2003-04-01.

[8] Object Management Group. *XML Metadata Interchange (XMI) Specification, Version 2.0*, May 2003. formal/2003-05-03.

[9] T. Rötschke. Adding pluggable meta models to FUJABA. In *Proc. Fujaba Days 2004*, 2004. To appear.

The TopModL Initiative

Pierre-Alain Muller
pa.muller@uha.fr
INRIA/Irisa – Université de Rennes – France

Cédric Dumoulin
cedric.dumoulin@lifl.fr
LIFL – Université de Lille – France

Frédéric Fondement
frederic.fondement@epfl.ch
EPFL/IC/LGL – Lausanne – Switzerland

Michel Hassenforder
m.hassenforder@uha.fr
MIPS/LSI – Université de Haute-Alsace – France

Abstract

We believe that there is a very strong need for an environment to support research and experiments on model-driven engineering. Therefore we have started the TopModL project, an open-source initiative, with the goal of building a development community to provide:

- an executable environment for quick and easy experimentation,
- a set of source files and production line,
- a web portal to share artefacts developed by the community.

The aim of TopModL is to help the model-engineering research community by providing the quickest path between a research idea and a running prototype. In addition, we also want to identify all the possible contributions, understand how to make it easy to integrate existing components, while maintaining architectural integrity. At the time of writing we have almost completed the bootstrap phase (known as Blackhole), which means that we can model TopModL and generate TopModL with TopModL.

Beyond this first phase, it is now of paramount importance to gather the best possible description of the requirements of the community involved in model-driven engineering to further develop TopModL, and also to make sure that we are able to reuse or federate existing efforts or goodwill.

This paper is more intended to set up a basis for a constructive discussion than to offer definitive answers and closed solutions.

Introduction – About model-driven engineering

At the end of the year 2000, the OMG proposed a radical move from object composition to model transformation[1], and started to promote MDA[2] (Model-Driven Architecture) a model-driven engineering framework to manipulate both PIMs (Platform Independent Models) and PSMs (Platform Specific Models). The OMG also defined a four level meta-modeling architecture, and UML was elected to play a key role in this architecture, being both a general purpose modeling language, and (for its core part) a language to define metamodels. As MDA will become mainstream, more and more specific metamodels will have to be defined, to address domain specific modeling requirements. Examples of such metamodels are CWM (Common Warehouse Metamodel) and SPEM (Software Process Engineering Metamodel). It is likely that MDA will be applied to a wide range of different domains.

While preeminent in the current days, MDA is only a specific case, and we suggest considering model-driven engineering as a wider research field, which includes the study of the following issues:

- What are the essential entities for model-driven engineering?
- How to classify these entities?
- How to translate models into executable code?
- What are the essential operations of model-driven engineering?
- How to classify these operations?
- How to separate and merge the business and platform aspects?
- How to build transformation systems?
- How to maintain a model-driven application?
- How to migrate a legacy application to a model-driven application?
- How to integrate conventional application with model-driven applications?
- Which abstractions and notations should be used to support the previous points?
- What kind of supporting environment should be defined?

Obviously the scope of model-driven engineering is wide and a lot of work is still ahead of us. We believe that a common research platform which would provide the fundamentals services required by model-driven engineering would significantly contribute to the advance of research in this field.

Basic principles of model-driven engineering

The point is to identify the fundamental characteristics upon which to build model-driven engineering. In our specific case, this means identify the requirements for a supporting environment dedicated to model-driven experiments.

The fundamental principles identified so far by the TopModL initiative are:

- The fact that everything is a model. For TopModL, models are first-class entities; everything is expressed explicitly in terms of models, including business models, platform models, executable models, debugging models, trace models, transformation models, process models…

- The notions of languages, models and roles. A model is expressed in a language; this language is a model which plays the role of meta-model for the models expressed in that language.

- The fact that TopModL itself is a model. We want TopModL to be model-driven; we want everything in TopModL to be explicit and customizable.

- The fact that everything is explicit, including the meta-modeling framework. For instance TopModL does not require the M3 to be MOF, it does not even require a 4 layer meta-modelling architecture.

- The independence versus the model repository. We want to have a uniform access to several repositories including EMF[3], MDR[4], or XDE.

These principles shape the requirements for the services to be provided by TopModL.

Toward a research platform for Model-Driven Engineering

The goal of TopModL is to provide an infrastructure for model-driven engineering, including a reference implementation of MDA as promoted by the OMG. TopModL wants to act as a facilitator for researchers in model-driven engineering in connection with other fields including generative programming, graph transformations, domain specific languages (DSLs), or aspects.

The basic services offered by TopModL are:

- Model (meta-model) persistence
- Model (meta-model) serialization in XMI (XML Meta-data Interface)
- JMI (Java Meta-Data) interfaces generation for model (meta-model) manipulation
- Visual edition of models (meta-models)
- Model-Driven parameterization of TopModL
- Model-Driven textual editor generation
- Model-Driven visual editor generation
- OCL evaluation during meta-model and model edition
- Code (Java, SQL) generation
- Model transformation

The TopModL artefacts include:

- A set of source files (Java, XML, text…) and model-driven production line to bootstrap TopModL.

- An executable environment. The first release (known as Blakhole) allows the visual edition of meta-model which conforms to the UML Infrastructure (TopModL by default uses the UML Infrastructure as M3).

- A model-driven web portal to share artefacts developed by the community (libraries of meta-models, profiles, models and transformations…).

The first release ot TopModL (Blackhole) and contains all the artefacts required to bootstrap TopModL. Bootstrapping means being able to use TopModL to model and generate TopModL itself.

Blackhole - Technical architecture of the bootstrap

The technical infrastructure of TopModL has been a recurring concern since the inception of the TopModL initiative (November 2003). The basic question is: *What should be the existing technology (if any) onto which to layer the new developments to be done in the initiative?*

The initial partners acknowledged that the answer to that question would not be obvious, and considered that trying to determine upfront the best technology would be counterproductive, and would significantly delay the timeliness delivery of the TopModL artefacts.

The decision was then taken to use the technology that the partners felt the most comfortable with at the time of the launch of the project, and not to wait for the next wonderful technology yet to come. The direct advantage of this approach was to accumulate practical experience and then to be able to make decisions based on explicit knowledge - rather than informal feelings - if a new technical element had to be incorporated.

The technologies and standards that were chosen for the bootstrap phase include:

- Java for the programming language
- JMI for the metadata API
- MDR for the model repository
- SWT for the graphical interface
- The Eclipse framework and plugging mechanism
- Eclipse for the IDE
- The UML Infrastructure as M3

It is the intent of the TopModL partners to take any appropriate action to simplify the potential transition from one technology to another one. Technology independence is achieved as much as possible by:

- Sticking to established standards when these standards are available (JMI, XMI, MOF, UML…)
- Defining neutral pivot on top of alternative technical solutions, for instance to be independent of the repositories.

The following picture describes the bootstrap process of TopModL.

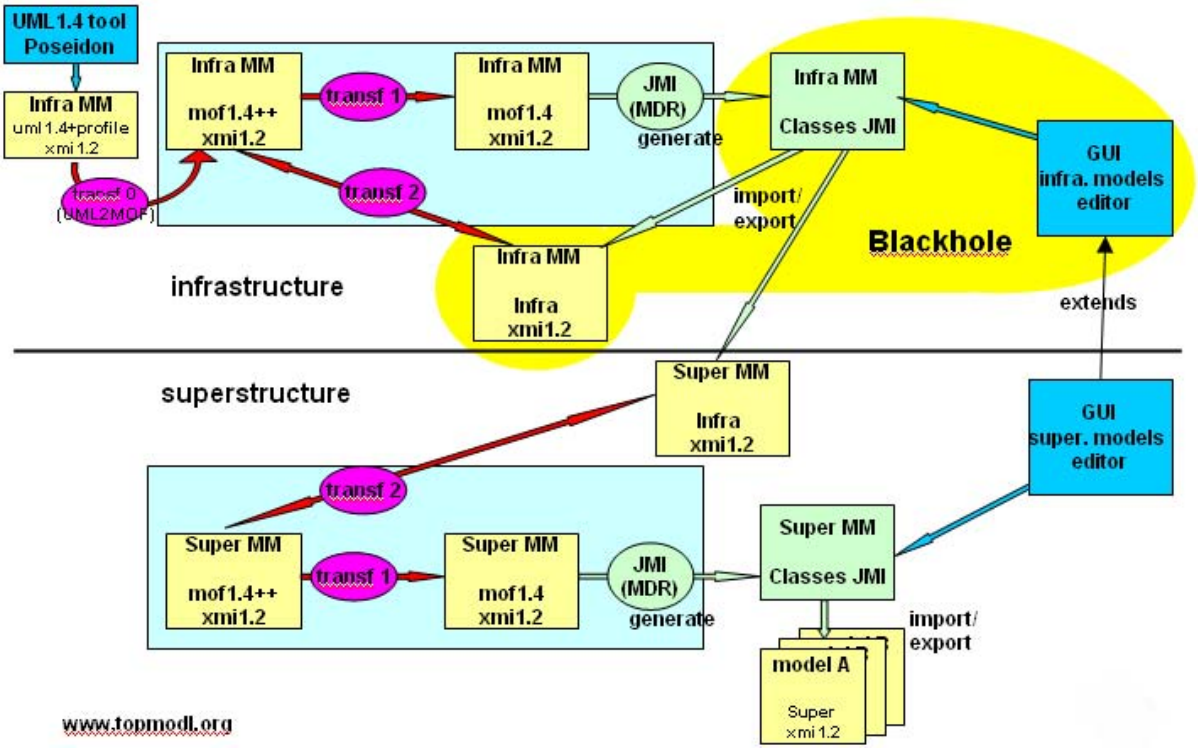

Figure 1 : Bootstrapping TopModL

We start by modelling our M3 layer (currently UML Infrastructure, but could be MOF or other) with the community edition of Poseidon[5]. We use UML 1.4 and a MOF 1.4 profile, and then promote the UML model of the Infrastructure to a MOF model using the MDR utilities UML2MOF. We were not completely able to express the Infrastructure with MOF this way (because of features like the package merge) and so we had to workaround by modifying the XMI files by hand (which in essence means that we are encoding the Infrastructure into some temporary extension of MOF that we name MOF1.4++). Then, we use a model transformation (transf 1) to translate the Infrastructure into a MOF 1.4 model which can be fed into the JMI/MDR generation process, and get a repository for Infrastructure based models (plus XMI serialization). We have developed a visual editor which connects to this repository, and we can this way edit metamodels (which conforms to UML Infrastructure). This editor is currently hand-coded, but it is our intent to generate it from models, in a similar way of what is doing Netsilon[6].

Another bi-directional model transformation (Transf 2) makes it possible to upload the Infrastructure model which was created with Poseidon. This transformation is then also used to Bootstrap TopModL, as it is now possible to edit the Infrastructure model with the TopModL visual editor, and then feed the generation process previously used to generate the repository.

The second part of the picture shows how the approach can then be used to generate a tool for the UML superstructure. The following paragraph further motivates such UML CASE tool generation.

Toward a model-driven CASE tool for the UML

TopModL is a meta-environment which can be used to realize model-driven CASE tools. The TopModL development community has considered that realizing a model-driven UML CASE tool would be an excellent use case for TopModL, and would also allow bypassing the current limitations of the available tools, most notably:

- The fact that no commercial tool fully implements the OMG standards, mainly because the first generation of UML CASE tools do not rely on an explicit description of the meta-models, but on hand-coded implementations, often themselves derived from earlier modelling tools (designed before the advent of the UML). Our experience also shows that CASE tools vendors do not have the resources to realize quick update of their tools, or to satisfy specific demands (to validate a research advance or to support a specific extension, or even simply to implement a standard).

- The lack of model-driven open-source tools, mainly because all the open-source efforts are based on programming and not modelling. TopModL is an open-model project before being an open-source project.

- The hard-wired nature of existing tools which embed a lot of decisions in their code and are therefore very difficult to customize. When buried in the code, decisions are implicit, as opposed to explicit when they are expressed in models and meta-models

In this context, TopModL will develop a new kind of UML CASE tool, entirely model-driven, with all the decisions made explicit by means of models. It is more accurate to talk about a product line, as the resulting CASE tool will always be conforming to a user-defined set of meta-models which will define its context. These capacities of customization will concern:

- The internal repository, generated in conformance to the meta-models referenced by the context.

- The model serialization (import-export via XMI)

- The visual editors, derived from a generic graph editor by explicit customization (this feature will be of special interest to realize domain specific languages).

- The textual editors (with syntaxic and semantic completion) generated from meta-models (the abstract syntax) and mapped on a concrete syntax (also expressed via models).

- The overall behaviour of the tool which will be described in a process model conforming to SPEM (Software Process Engineering Model).

Related works

There are many related works, which share a common vision with the TopModL initiative. We summarize some of these approaches below:

- Meta CASE tools including Metaedit+[7], Dome[8] or GME[9], provide customizable CASE tools, however they are fairly closed in the sense that they are neither open-source, nor themselves model-driven.

- Dedicated model-driven tools, which generate specific applications, like Netsilon[6] for Web information systems, Accord/UML[10] for embedded distributed real-time systems.

- OCL based tools, including KMF[11] which generates modelling tools from the definition of modelling languages expressed as meta-models, or Octopus[12] an Eclipse plug-in OCLE[13] or the Dresden OCL toolkit[14], which are able to check the syntax of OCL expressions, as well as the types and correct use of model elements like association roles and attributes.

- Meta-modelling frameworks like Eclipse EMF[3], Netbeans MDR[4] or Coral[15] which offer model persistence, model serialization and programmatic access to models via an API, , or integration technologies like ModelBus[16]. These frameworks provide part of the functionalities required by TopModL.

- Open-source modelling tools, including ArgoUML[17] of Fujuba[18] which offer significant feature at the M1 level, but lack customization at the M2 level.

One of the goals of TopModL is to understand how to reuse or leverage these related works, and to find how to integrate them as much as possible in a research platform for model-driven engineering.

Conclusion

The TopModL open-source initiative has been launched with the goal of providing tool support to the model-driven engineering research community.

The TopModL initiative groups a development community, a web portal to share the artefacts developed by the community, a set of source files and an executable program for meta-modeling.

TopModL is itself a model-driven application, and a first phase known as Blackhole delivers the bootstrap of TopModL, which means that TopModL is modelled and generated with TopModL.

References

[1] J. Bézivin, *"From Object Composition to Model Transformation with the MDA"*, in proceedings of TOOLS'2001. IEEE Press Tools#39, pp. 350-354 . (August 2001).

[2] Object Management Group, Inc., *"MDA Guide 1.0.1"*, omg/2003-06-01, June 2003.

[3] Eclipse EMF, web site http://www.eclipse.org/emf/

[4] Netbeans MDR, web site http://mdr.netbeans.org/

[5] Poseidon web site http://www.gentleware.de

[6] P.-A. Muller, P. Studer, and J. Bezivin, "*Platform Independent Web Application Modeling"*, in P. Stevens et al. (Eds): UML 2003, LNCS 2863, pp. 220-233, 2003.

[7] R. Pohjonen, "Boosting Embedded Systems Development with Domain-Specific Modeling", in RTC Magazine, April, pp. 57-61, 2003

[8] Honeywell, 1992, "*DOME Guide"*, available from "www.htc.honeywell.com/dome/"

[9] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, "*The Generic Modeling Environment"*, Proceedings of the IEEE International Workshop on Intelligent Signal Processing, WISP'2001, Budapest, Hungary, may 24-25, 2001

[10] S. Gérard, N. S. Voros, C. Koulamas, and F. Terrier, "*Efficient System Modeling of Complex Real-Time Industrial Networks Using the ACCORD UML Methodology*", DIPES'2000,Paderborn, Germany, 2000.

[11] Kent Metamodeling Framework, web site http://www.cs.kent.ac.uk/projects/kmf/index.html

[12] Octopus web site http://www.klasse.nl/ocl/octopus-intro.html

[13] OCLE web site http://lci.cs.ubbcluj.ro/ocle/

[14] Dresden OCL toolkit web site http://dresden-ocl.sourceforge.net/

[15] Coral, web site http://mde.abo.fi/tools/Coral/

[16] X. Blanc, M.-P. Gervais, P. Sriplakich, "Model Bus : Towards the interoperability of modelling tools", MDAFA'04, Linköping, June 10-11, 2004

[17] ArgoUML web site http://argouml.tigris.org/

[18] Fujaba web site http://wwwcs.upb.de/cs/fujaba/index.html

Adding Pluggable Meta Models to *FUJABA*

Tobias Rötschke
Fachgebiet Echtzeitsysteme
Institut für Datentechnik (FB18)
Technische Universität Darmstadt
Darmstadt, Germany
tobias.roetschke@es.tu-darmstadt.de

ABSTRACT

In this paper we propose to split the structural part of the Fujaba meta model in an internal, tool-specific and and external, standardized metamodel. External metamodels and adequate editors (and code generators) should be provided by plugins. By this means, Fujaba would gain flexibility with respect to evolving meta model standards and allow the Fujaba community to reuse the Fujaba graph rewriting engine for different schema definition languages. In our opinion, this proposal is a major contribution to strengthen the Fujaba community and keep the various development streams together.

1. INTRODUCTION

One of the most prominent features of Fujaba[21] is the ability to visually specify behaviour by means of graph transformations written in the *Story Driven Modelling (SDM)*[6] language. Unlike its predecessor PROGRES [19], Fujaba aimed for adopting standard modelling languages which are well-known to a large community of software engineers. Consequently, Fujaba uses a fixed UML-like meta model, which is referred to throughout the system. Graph schemata are defined by UML 1.x class diagrams, transformation are specified with slightly modified UML activity diagrams, collaboration diagrams, and state charts.

However, as new languages like MOF 2.0 [14] and UML 2.0 [16] come up, Fujaba should be able to evolve with these languages to stay competitive and hence avoiding one of the major drawbacks of PROGRES. The basic idea would be to separate the internal SDM meta model required for the graph rewriting engine from an external meta model to allow the user to specify in the modelling language of his choice. While the internal model belongs to the FUJABA core, the external meta model would be added using the FUJABA plugin mechanism.

During our ongoing effort to add a MOF 2.0 schema editor with JMI[5]-compliant code generation while still using SDM for behavioural specifications [1], we found it necessary to perform this separation and successfully managed to implement some critical parts of it. Our current FUJABA implementation allows the user to choose between the original UML 1.x meta model and a new MOF 2.0 meta model, edit class diagrams and generate appropriate code.

However, adopting our version would require plugin developers to adapt their plugins. As we try to stay as close to the original implementation as possible, the effort will be basically reduced to renaming meta model related type references. In our opinion the benefits of the proposed modification will clearly outweigh this inconvenience. So we hope that the meta model separation will be re-integrated in the main branch after the Fujaba Days.

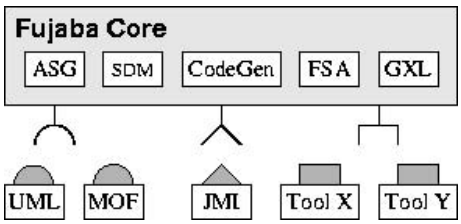

Figure 1: Our Fujaba vision

Figure 1 scetches our Fujaba vision. The Fujaba core contains all generic packages like *ASG*, *FSA*, *CodeGen*, and finally the internal *SDM* meta model. External meta models like *UML 2.0* or *MOF 2.0*, standardized code generators like *JMI* and – as before – various tools based on the Fujaba platform can be plugged into the system.

Section 2 describes the shortcomings of the current meta model. In section 3, we propose how FUJABA should be reengineered to be able to support pluggable meta models and code generators.

2. ABOUT THE FUJABA META MODEL

As discussed in section 1, Fujaba has a fixed meta model and code generator, which cannot be replaced easily. Besided, Fujaba lacks support for a standard application interface for (meta) modelling tools, for instance JMI. This has also been criticized in [18], where accordingly a new refactoring tool called JMI Conform Model Transformator Generator (JCMTG) is derived from Fujaba using a MOF 1.4 meta model [13] and the AndroMDA [3] pluggable code generator. In [1], another project is described, where Fujaba is adapted to a MOF 2.0 meta model and the MomoC code generator [2] is used. However, creating and maintaining a new tool for every meta model would not be wise, since there is a significant overlap between meta models and little variety with respect to SDM requirements. Instead it would be better to allow pluggable external meta models and code generators which are plugged into the Fujaba core using its own internal meta model.

In an ideal world, these features would have been part of the plugin mechanism introduced with version 4.0. Fujaba's key competence lies graph transformation engine, which distinguishes it from most other tool platforms. The current plugin mechanism allows only to build graphical editors based on the Fujaba meta model and code generator. But the true added value lies in application of the graph transformation engine to arbitrary meta models and code generators.

Apart from the above, presentation-related concepts of the Fujaba meta model like "project", "diagram" and "file" cannot be found in standard modelling languages like UML 1.x, UML 2.0 or MOF 2.0. In the UML 2.0 Diagram Interchange Specification [15] however, a meta model extension to UML is proposed, which allows to model presentation information. Unfortunately, it does not yet provide full support for UML 2.0 or MOF 2.0 and the document appears to be rather unfinished, although marked as "Final Adopted Specification". With respect to Fujaba, there should be a clear separation between standardized meta model elements and not yet standardized presentation elements. This would make it easier to adopt a designated standard early.

Looking into Fujaba in more depth, one finds that some classes are real god classes [17]. *UMLProject* for instance has almost 3000 lines of code and performs many tasks: Static methods for accessing administrative classes and performing string operations, complex algorithms for loading projects and finally, it acts as part of the Fujaba meta model. Accordingly, UMLProject is referred to throughout the whole system. Changing the meta model would hence result in countless modifications of the code.

Another drawback of UML Project is, that it only allows for one project instance. Especially when using Fujaba for model integration [7] or tool integration [4] purposes, it would make sense to work with multiple projects of different kinds. There would be one project for each integrated model or tool, which would manage its specific meta model, implements a specific algorithm to load or store model and so on. The integration part with its integration model should be an additional project depending on the model / tool specific projects.

Besides, the Fujaba meta model has some extensions for project definition and Java language artifacts, especially modifiers. Static classes [10], package visibility, final, native or synchronized methods are just a few examples. Apart from programming language specific visibilities [12], these artifacts are neither found in one of the official UML-meta models nor in the MOF meta models.

The conclusion is that the existing Fujaba meta model merges modelling language, programming language and project concepts into a single meta model. Assuming for the time being, that programming language and project concepts are fixed within the Fujaba context, it should still be possible to use different meta models for the modelling language, because evaluating and improving evolving modelling languages is one of the major research challenges that should be addressed by the Fujaba community.

Many features like behavioural diagram editors and the code generator are related to the existing all-in-one meta model. Our idea is to separate between an internal so-called SDM meta model and an external meta model, which could be UML 1.x, UML 2.0, MOF 2.0 or similar. As we want to *replace* parts of the meta model rather than *extend* it, the Fujaba plugin mechanism alone is not sufficient. On the one hand we would have to duplicate much of the Fujaba code to reuse the existing Fujaba platform when adding an extra meta model, and just writing another editor plugin for it. This would hamper Fujaba's maintainability. Extending the Fujaba meta model by means of inheritance on the other hand, would also not solve the problem, as most meta models obviously are not a superset of UML 1.4. Apart from the threat of name collisions, this would result in overly large classes and needlessly increase the runtime-overhead.

3. PROPOSED SOLUTION

This section describes, how we propose to reengineer the Fujaba core, so that different meta model plugins could be used with the Fujaba platform. First we describe, how UMLProject has to be refactored to allow for multiple, meta model-specific projects in a Fujaba session. Next, we present the designated internal Fujaba meta model before we explain, how an external meta model is linked to the internal meta model. Finally, we describe how plugin developers have to adjust their code to make the plugins work with the new Fujaba core.

3.1 Breaking up UMLProject

In our implementation, *UMLProject* is replaced by four new classes: ProjectManager, ProjectLoader, SDMMetaModel, and SDMProject. Figure 2 shows the interaction of these classes.

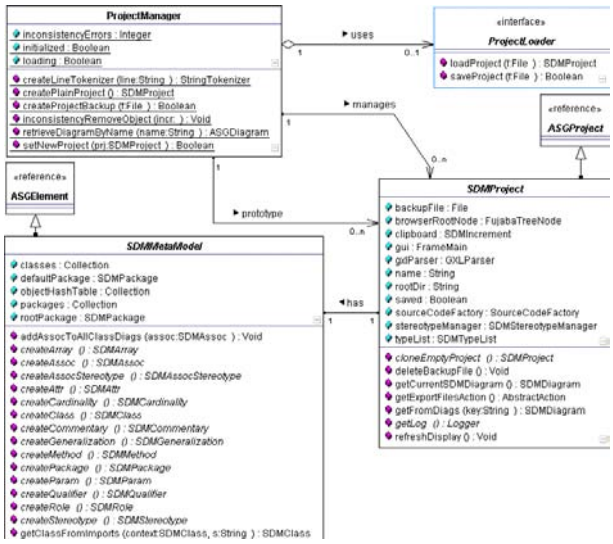

Figure 2: Refactored UMLProject

The *ProjectManager* takes over the static part of the former UMLProject. It is responsible for managing instances of projects, meta models, and loaders. The *ProjectLoader* is responsible for loading and storing Fujaba models. Each external meta model will usually have its own loader, but even different loaders could be combined with a meta model

(e.g. XMI, GXL, FPR, or native tool specific loaders) following the *Strategy* [8] pattern. *SDMMetaModel* is responsible for creation of and access to model element instances. Most methods are abstract, as the concrete meta model plugin decides, how instances are managed. Concrete methods usually implement query algorithms or map model elements to Java entities, which is considered being part of the internal meta model. Finally *SDMProject* contains the purely project related stuff concerning diagrams, import/export and filesystem entities, which is not covered by an external meta model.

3.2 The internal meta model

Fujaba's meta model is geared to UML 1.5, but contains several Java-specific and project-related extensions as discussed in section 2. To be able to replace the standardized part of the meta model by a newer variant, we distinguish between an internal and an external meta model. The external meta model is a standardized UML or MOF model without any tool specific extensions. This could be UML 1.5 to stay backwards compatible with existing Fujaba plugins, but also UML 2.0 or MOF 2.0. Apart from some renaming operations, the new Fujaba core together with the UML 1.5 meta model plugin will provide full backward compatibility for tool plugins.

The internal meta model contains abstract methods for all features that Fujaba expects from the external meta model. All project or Java-related features are already implemented in the internal meta model as they are out of the scope of the external meta model. Figure 3 illustrates the new design, using a slightly simplified definition of *SDMClass*.

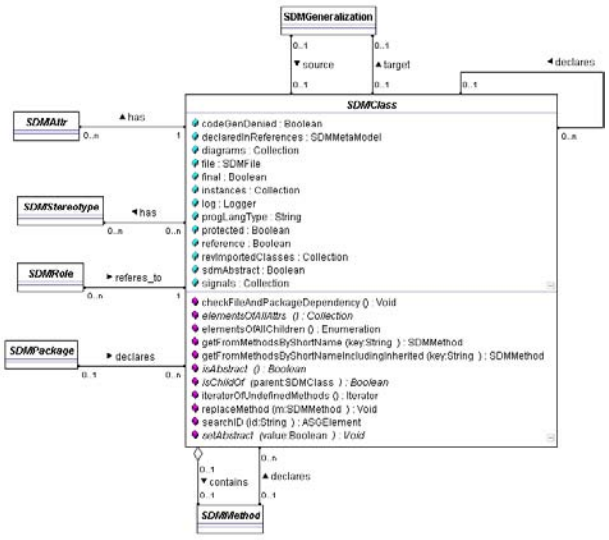

Figure 3: Example element of the SDM meta model

All presented attributes and concrete methods are considered being part of the internal Fujaba model. Each attribute represents not only the attribute declaration, but also the usual access methods generated by the Fujaba code generator[1]. All presented associations and abstract methods have to be realized by the external meta model implementation.

We call the internal meta model *SDM* rather than *Abstract* or *Common* metamodel, as it does not only contain the structural but also the behavioural part of the Fujaba metamodel. But only the structural part is refined by the external meta model.

3.3 Integration of meta model plugins

To implement a plugin for an external meta model, the model element implementations must be subtypes of the corresponding SDM element. In case of the UML 1.5 meta model this is trivial. The new element *UMLClass* for instance, would just specialize *SDMClass* and implement all abstract methods as before. Only the minor changes for plugin development described in section 3.4 do apply.

For other meta model implementations, like our JMI-implementation of the MOF 2.0 metamodel, the meta model-specific interface must be adapted to the SDM metamodel. Two aspects are important to be able to reuse the Fujaba platform: Every model element has to be a subtype of ASGElement, or its corresponding SDM element to be more precise. Besides, every modification to the model must probably invoke *firePropertyChange* with adequate parameters to perform incremental updates of the presentation.

Method calls to the SDM interface must be delegated to the corresponding meta-model specific method calls. This situation has been described in [8] as *Adapter* design pattern. Using the class adapter[2] variant of this pattern, only one runtime instance exists per model element instance, and attributes need not to be duplicated. This keeps the overhead low. Figure 4 demonstrates the adapter pattern for the plugin class *MOFClass* conceptually.

MOFClass is the JMI-interface of the MOF 2.0 class concept. It correspondes to the internal *SDMClass* concept. The MomoC[2] code generator provides a JMI-compliant default implementation *MOFClassImpl*, which is conceptually independant from the tool. The class *FujabaMOFClassAdapter* is a hand-written class, which adapts the MOF implementation to the Fujaba tool platform. However, as Java does not provide multiple inheritance, the generalization relationship *FujabaMOFClassAdapter* to *SDMClass* has to be moved up to *MOFClassImpl*. This is easily done by adjusting the MomoC templates.

Both *MOFClass* and *SDMClass* provide an attribute *abstract*, which is accessed by identically named access methods called `setAbstract`. In *MOFClassImpl*, this method is implemented without being aware of the Fujaba platform. In *FujabaMOFClassAdapter*, this implementation is overridden by the following code:

```
void setAbstract (boolean value) {
  boolean changed = false;
  if (super.isAbstract() != value) {
    super.setAbstract(value);
```

[1] e. g. `addToX`, `removeFromX`, `iteratorOfX`, `elementsOfX`, `sizeOfX`, `hasInX`, `removeAllFromX` for a set valued association end 'x'

[2] as opposed to the object adapter pattern which is used for FSA, where every FSA object creates an additional Swing object

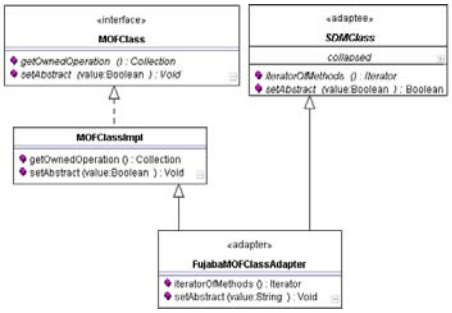

Figure 4: Applying the class adapter pattern

```
    changed = true;
    firePropertyChange ("abstract", !value, value);
  }
   // return changed;
}
```

This example demonstrates, how generated and handwritten code can be combined nicely. But there is also a naming conflict as the return types of the `setAbstract` method in *SDMClass* and *MOFClass* do not match, while the signature is identical. We are currently working on systematic prefix conventions for the internal meta model to allow external meta models to be as close to the standard as possible. Covariant returns[9] as proposed for the next Java version, would provide a clean solution to this problem.

The following code fragment shows, how external and internal meta model can be connected, even if there are minor conceptual differences. However, attention must be paid to the correct usage of Fujaba *properties* as reflection occurs at many occasions. So when implementing the unparse module for *MOFClass*, the *ClassCompartmentVisibilityUpdater* for MOF operations must be initialized with the property "methods", because the invocation of `iteratorOfMethods` is triggered by this name via reflection.

```
Iterator iteratorOfMethods {
  return super.getOwnedOperation().iterator();
}
```

Apart from the above, the internal and the external meta model do not always fit perfectly together. Some thought has to be put on the mapping of *MOFAssociation* to *SDMAssoc* as MOF associations are far more powerful than former UML associations. But as most of the MOF-specific functionality is hidden in the code generated by the MomoC tool, adaption should be possible. Somewhat more inconvenient is the fact that an instance of *MOFProperty* corresponds to instances of *SDMAttr*, *SDMRole* and *SDMCardinality*. In this case, we use the Object Adapter pattern [8] rather than the Class Adapter pattern, which results in slightly more runtime objects. *MOFElementImport*, which is used to create references to elements from foreign *MOFPackage* instances, acts as *SDMClass* or *SDMAssoc* from the Fujaba point of view and can be realized as object adapter as well.

3.4 Impact on other plugins

The increased flexibility of our proposal comes at the cost of some refactoring effort for plugin developers. Fortunately, the modifications are rather trivial.

As the Fujaba core does not contain a concrete meta model implementation anymore, most plugins will depend on a meta model plugin like the UML- or MOF-plugin. As *UMLProject* has been redesigned, calls of `UMLProject.get()` have to be replaced by `ProjectManager.getProject(...)`. Although not yet implemented, it would be easily possible to allow the ProjectManager to handle multiple projects, each depending on a different meta model plugin. References to the internal Fujaba meta model using `UML...` have to be renamed into `SDM...`. The prefix "UML" would refer to an external meta model imported by the UML-Plugin. When creating a new instance of a model element inside the core, it is no longer allowed to directly call the constructor of this element. Instead of calling for instance

```
clazz = new UMLClass(name);
```

one would have to call

```
SDMProject project = ProjectManager.getProject();
SDMMetaModel model = project.getMetaModel();
clazz = model.createClass();
clazz.setName (name);
```

This might look more complicated initially, but SDMProject and SDMMetaModel are referred to very often throughout the code and hence readily available. In practice, only a few extra lines have to be added. For the sake of backward compatibility, the UML meta model plugin will still provide the old constructors, so that existing tool plugins are easily adapted.

4. CONCLUSION

In this paper we have proposed, how to split the Fujaba meta model into an internal SDM meta model and standard-compliant external meta models that can be provided by plugins. We actually consider this a logical and necessary step, which should already have been implemented by the plugin mechanism introduced quite recently. We have pointed out that the Fujaba community will benefit from keeping up with evolving meta model specifications, without the necessity to modify the core. Considering our current needs, external meta models have been restricted to structural diagrams, while behavioural diagrams are reused based on the internal meta model.

We are currently implementing the proposed changes in the "Refactoring" branch of the Fujaba CVS repository and prepare a demo of the modified Fujaba core together with a UML and a MOF plugin. Our effort is driven by the desire to use Fujaba as MOF 2.0 editor and JMI code generator. Based on the original Fujaba core, our goal could not be reached, as the Fujaba code generator which has been used to bootstrap Fujaba violates JMI guidelines beyond repair. Similar problems have already forced the JCMTG project

to separate from the Fujaba development stream. We are convinced that our proposed changes are necessary to keep the Fujaba community together and improve the maintainability of the code base.

From our experiences with transforming the Fujaba core and adapting the MOF plugin, we estimate the migration effort for plugin developers to be approximately one or two days. Some additional effort would be required to integrate all modifications to the main branch since the refactoring branch has been created. The sooner both branches are merged the better. Together with the UML-Plugin, the refactoring branch should provide the same features as the main branch.

Currently, their are only two major concerns: On the one hand, Java lacks multiple inheritance between classes, resulting in either more runtime objects or code duplication. On the other, Fujaba and MOF/JMI use different approaches for association implementation, which are not easily coupled.

The ideas presented here would allow to use the Fujaba code generator for different meta models. Consequently, we will investigate how pluggable code generators (e.g. JMI compliant) can be realized based on the proposed modifications. As the existing code generation concepts are already rather flexible, we are confident that pluggable code generators can be realized with reasonable effort.

5. REFERENCES

[1] C. Amelunxen. MOF 2.0 Editor Plugin for Fujaba. In Schürr and Zündorf [20]. To appear.

[2] L. Bichler. Tool Support for Generating Implementations of MOF-based Modeling Languages. In J. Gray, J.-P. Tolvanen, and M. Rossi, editors, *Proceedings of the 3rd OOPSLA Workshop on Domain-Specific Modeling*, Anaheim, California, USA, October 2003.

[3] M. Bohlen. *AndroMDA: From UML to deployable components*, 2004. http://www.andromda.org.

[4] S. Burmester, H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. In *Proc. Workshop on Tool Integration in System Development*, Helsinki, Finland, September 2003.

[5] R. Dirckze. *Java Metadata Interface (JMI) Specification, v1.0*. Unisys Corporation, Sun Microsystems, Inc., June 2002. http://java.sun.com/products/jmi/.

[6] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Grammar Language based on the Unified Modelling Language and Java. In *Workshop on Theory and Application of Graph Transformation (TAGT'98)*. University-GH Paderborn, Nov. 1998.

[7] R. Freude and A. Königs. Tool Integration with Consistency Relations and their Visualization. In *Proc. Workshop on Tool Integration in System Development*, Helsinki, Finland, September 2003.

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

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, 3rd edition draft*, 2004.

[10] JavaWorld. Static class declarations, 1999. http://www.javaworld.com/javaqa/1999-08/01-qa-static2.html.

[11] M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes on Computer Science*. Springer, 1996.

[12] Object Management Group, Inc. *OMG Unified Modeling Language Specification Version 1.4*, Sept. 2001. http://www.omg.org/docs/formal/01-09-67.pdf.

[13] Object Management Group, Inc. *Meta-Object Facility (MOF) Specification Version 1.4*, Apr. 2002. http://www.omg.org/docs/formal/02-04-03.pdf.

[14] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Specification*, Oct. 2003. http://www.omg.org/docs/ptc/03-10-04.pdf.

[15] Object Management Group, Inc. *UML 2.0 Diagram Interchange Specification*, Sept. 2003. http://www.omg.org/docs/ptc/03-09-01.pdf.

[16] Object Management Group, Inc. *Unified Modeling Language (UML) Specification: Infrastructure Version 2.0*, Sept. 2003. http://www.omg.org/docs/ptc/03-09-15.pdf.

[17] A. J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[18] H. Schippers. JMI Conforme Modeltransformator-Generator. Master's thesis, Universiteit Antwerpen, 2004. In Flemish.

[19] A. Schürr, A. J. Winter, and A. Zündorf. Developing Tools with the PROGRES Environment. In Nagl [11], pages 356–369.

[20] A. Schürr and A. Zündorf, editors. *Fujaba Days 2004*. TU Darmstadt, 2004. Technical Report. To appear.

[21] A. Zündorf. *Rigorous Object Oriented Software Development*. Universität Paderborn, 2001. Habilitation Thesis.

FASEL: Scripted Backtracking for Fujaba

Erhard Schultchen
erhard@i3.informatik.rwth-aachen.de

Boris Böhlen
boehlen@cs.rwth-aachen.de

Ulrike Ranger
ranger@i3.informatik.rwth-aachen.de

Department of Computer Science III, RWTH Aachen University
Ahornstr. 55, 52074 Aachen, Germany

ABSTRACT

We implemented a new scripting language for Java, which is especially useful for applications based on code generated by Fujaba. Besides simple control structures also found in other scripting languages, this new approach allows nondeterministic execution using backtracking. This is demonstrated by means of the Ferryman Problem, which is a well known example for nondeterministic graph rewriting systems.

1. INTRODUCTION

In an earlier work, we developed a graph-based tool called "eHome-Configurator" that is based on a data model generated with the help of Fujaba [4]. The term "eHome" denotes a home which offers its inhabitants advanced benefits through the combination of its electronical equipment.

The eHomeConfigurator was designed to operate on the created data model and thus enabling the user to apply a given "eHome scenario" onto an environment (a house or an apartment). Here, scenarios are modeled and displayed as a tree of functional requirements. It is possible to model a "light-control scenario" by adding requirements for a controller device to switch lights in every room on and off. Furthermore, a movement detection has to be added, so the eHome can determine where the customer currently resides. An environment is modeled by means of locations (like rooms) connected and interconnected by location elements (doors, windows , ...) as well as all installed devices.

The user's task is to match already installed devices in the customer's household to requirements demanded by the chosen scenario. Requirements that cannot be fulfilled by installed devices require the installation of new devices. In our realization, the user calls graph rewriting rules, which are specified using Fujaba and are called "Activities" or "Story Diagrams". These rules imply certain constraints. For example, devices may only be connected to other devices with a compatible interface. Furthermore, rules determine whether all specified requirements are fulfilled.

During the development of the eHomeConfigurator, the idea arose to specify scenarios not only in an interactive way by using the tool itself, but also to create scenarios by a "script" which uses graph rewriting rules. As these rules are transformed to Java methods, a script has to be able to call arbitrary Java methods, pass appropriate parameters and interpret the return value. This approach separates the specification of a scenario from its application to a given environment. In addition, scenarios can be loaded dynamically into the tool, just by running the created script.

We developed a simple scripting language named "Fasel" which is short for "Fujaba Activity Scripting Environment Language". It is designed for the use in applications that are built upon code generated with the help of Fujaba. Fasel can be used for arbitrary Java applications as well.

Furthermore, we modified the eHomeConfigurator mentioned above to solve the Ferryman Problem. Therefore, a new Fujaba model had to be specified, formalizing the elements and rules for this problem. Finally, Java code was generated for the new model and was integrated into the eHomeConfigurator. Also the eHomeConfigurator had to be adapted to the new data model, but this only required few efforts as the eHomeConfigurator was designed in an abstract and generic way. The modified tool is called "Ferryman Configurator". In section 2 we use the Ferryman Configurator to demonstrate the power of our scripting language. Section 3 contains a more elaborated introduction to Fasel's syntactical elements. We conclude in section 4 with an overview on current development and further possibilities

2. USAGE EXAMPLE

To emphasize the usefulness of the presented scripting language, we implemented the Ferryman Problem in terms of a graph model using Fujaba. To solve this problem, three things have to be taken across a river: A wolf, a sheep, and a cabbage. The ferry available may only take one cargo besides the ferryman, who has to row the ferry. Leaving either wolf and sheep or sheep and cabbage alone on one riverside is not allowed, as the first would eat up the second.

Our goal is to solve this problem automatically. We built an application that uses the Java code generated for this problem, in order to load, unload, and move the ferry from one riverside to the other (figure 1). There are methods to check whether the current situation is allowed or the problem has already been solved. What is missing is some kind of a solving algorithm, which could be formalized as follows:

At each step, choose a cargo from the current riverside and take it to the other side. The ferryman may also choose to go empty. The

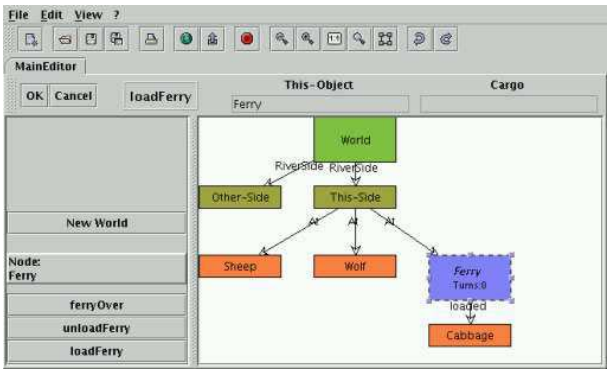

Figure 1: Ferryman Application

situation has to be allowed with regard to cargos left together on a riverside. Continue until the problem is solved.

In this section, we present the script to solve the Ferryman Problem. A more detailed introduction to Fasel's syntax follows in section 3.

2.1 Nondeterminism and Backtracking

A severe problem about Fujaba is its way to find correct nodes for a certain activity: Once a story pattern is executed, changes made to the graph are persistent. This means it is not possible to revoke prior state whithout human intervention (runtime systems like Dobs [1] offer undo/redo functionality triggered by the user). The Ferryman problem requires such an undo facility: If a situation is not allowed, the last transformation has to be undone. Furthermore, the cargo chosen that lead to the disallowed situation has to be remembered for choosing a different cargo next time. If no alternative choice is left, then a prior step has to be revoked.

2.2 Implementation

The solving algorithm suggested above was implemented in Fasel (see figure 2 for line numbers). The main program (lines 28 to 38) first performs some initialization and sets variables for the programmer's conveinance (we can use *$F* instead of *$WORLDROOT.getFerry()*). Then, the *solve* procedure is called which will solve the problem recursivly (line 34).

The *solve* procedure first checks whether the ferry did too much turns - we introduced a maximum turn count of 7 in order to avoid endless loops (line 3). If more than 7 turns were already made, backtracking is initialized by returning `false`. Otherwise, if the problem has already been solved, execution is canceled and `true` is returned, causing all recursivly called procedures to return (line 6). Otherwise, *solve* tries to find a solution by transfering a cargo from the ferry's riverside to the other side.

The procedure *tryCargo* is used to transfer the cargo (lines 16 to 20) and to test whether the new situation is allowed (line 21). Moving the ferry and transfering the cargo is revoked in case of a prohibited state. The procedure quits and returns `false` in this case, causing the caller to try another cargo if available or to return `false` to its own caller. If this should run down to the main program, no solution for the problem could be found. Otherwise, if *tryCargo* conciders a given situation as safe, it will call *solve* recursivly to continue solving the problem from the new state (line 24). If this call returns `false`, another way has to be found and so the latest

```
1   proc solve()
      local $CARGO, $RES;
      if $F.getTurns() > $MAXTURNS then
        return false;
5     endif;
      if $W.finished() then
        return true;
      endif;
      return forany ($C in
10      [NONE | $F.getRiverside().iteratorOfCargo()];
        ) try :tryCargo($C);
    endproc;

    proc atomic tryCargo($C)
15    local $RES;
      if $C != NONE then
        $F.loadFerry($C);
      endif;
      $F.ferryOver();
20    $F.unloadFerry();
      if not $W.safeSituation() then
        revoke false;
      else
        decide :solve();
25    endif;
    endproc;

    package
      ["ggraph.model", "java.lang", "java.util"];
30  $FARS.iteratorType(SAFE);
    $MAXTURNS := num 7;
    $W := $WORLDROOT;
    $F := $W.getFerry();
    if :solve() then
35    write "Solved the ferryman problem";
    else
      write "No solution avaiable";
    endif;
```

Figure 2: Fasel Listing for Ferryman problem

turn done by *tryCargo* is revoked. The calling *solve* procedure has to find another cargo in this case.

If the recursivly called *solve* procedure returns `true`, the problem has been solved and *tryCargo* procedure returns `true` to its caller, causing it to do the same until the main program is reached.

3. FUJABA SCRIPTING LANGUAGE

In order to realize the basic idea to invoke Java methods by a script language, close interaction with the Java Virtual Machine is required. Fasel scripts are first transformed to a list of simple operations without nesting, using a compiler written in Java. This list may be stored as a "compiled" or "binary" form of the script, and recompiling a script on each run can be avoided. Operations are executed by the Fasel Runtime System called "FaRS". FaRS also manages the script's variable table and provides access to the data model.

FaRS itself is implemented in Java and runs in the host application's virtual machine (in our case, the Ferryman Configurator).

FaRS holds at least one root node to access the application's data model. For the Ferryman application presented in the previous section, FaRS holds the variable `$WORLDROOT` which denotes the single `World` node of this graph (see figure 1). From this node, all other nodes of the problem can be reached. For example, the world node is used to retrieve the ferry node in line 33, figure 2.

3.1 Syntax

Fasel's syntax is based on well-known imperative programming languages, such as Pascal. All variable names have to be preceeded by a "$" sign, statements are terminated by ";". Words not starting with a "$" are usually treated as a keyword if they match one, or as a constant string. Characters enclosed in quotation marks are treated as strings too.

":=" is used for assignments, following Pascal's syntax instead of C. Assignment statements require a variable on their left side and an arbitrary expression on their right side. It is possible to assign a "null" value which does not refer to any specific value at all. Fasel's null has the same meaning as Java's null.

Equality test is done by "==". The expression "`a == b`" is mapped to the Java expression `a.equals(b)`. Thus, "==" tests for equality of two objects instead of identity [2]. If the expressions on both sides of "==" yield a null value, the result is `true`. If only one expression is null, this test returns `false`.

Fasel offers common statements known to most imperative programming languages. Besides the `if ... then ... else ... endif;` statement, a `while ... do ... done;` loop is available. Fasel also contains a for-each loop: `for $V in ... do ... done;` For-each loops may iterate over generic collections (`java.util.Collection`) or use a given Iterator (`java.util.Iterator`). Also, Fasel supplies a direct list-expression to construct a list: `[ a, b, c, ... ]`. List expression are also used to concatenate lists by using the "|" character. The expression `[ a, b | $list ]` constructs a new list [a,b] and appends *`$list`*, assuming that *`$list`* contains a list.

To modify a list, Fasel provides the "`head $list`" and "`tail $list`" expressions. The first one provides the first element of the given list whereas the second returns a copy of the list with its first element removed. Both expressions are not applicable to empty lists or variables that do not contain a list. `head` and `tail` expressions are useful for recursive procedure calls often used in functional programming.

3.2 Activity Invokation

Activities, or arbitrary Java methods are called using the "`.`" operator. An object on the left side of the dot is used as the object to call the method from. If the left side is a constant string, a variable containing a string or a variable containing a `java.lang.Class` object, the method is called as static class method. The right side of the dot is taken as the method name. This may be given in form of a static string or as a variable containing a string.

Parameters are enclosed in paranthesis, like a regular java method. The method call may be part of an assignment statement, assigning the call's return value. If invoked methods throw an exception, an appropriate message is sent to the console, but execution of the script is continued. This behaviour is meant for simple batch processing where actitivities do not depend on each other and partially unsuccessful runs of a script are acceptable. However, if the script depends on the failed method's return value, a subsequent statement may find an unexpected `null` value and fail to continue.

Method call operators have left precedence, meaning if several method calls are placed in a single statement, the left-most call is performed first and its return value is used as the call object for the next one. This coincides with the Java syntax. For example, the following statement is handled just like in Java: `$obj.toString().substring(1).trim();`.

To invoke a method on a class rather than an object, usually the fully qualified classname has to be used. As Fasel currently does not distinguish between package- and class names, any expression containing a dot would be interpreted as a method call. Hence, fully qualified classnames have to be enclosed in quotation marks.

It is also possible to set a list of packages as the "package search list". The Fasel Runtime System browses this list whenever an unqualified classname is given. For example, calling `Integer.toString(int)` in Fasel would have to be written as *`''java.lang.Integer''`*`.toString(...)`. Setting the package search list using `package [` *`''java.lang.Integer''`* `];` allows to drop quotation marks and package qualifier and simplifies the call to `Integer.toString(...)`.

3.3 Subroutines

Fasel allows the declaration of subroutines, which are declared at the beginning of the file. No declaration before usage in terms of their order in the input file is required, in contrast to C. Subroutines are defined using the `proc` keyword (see figure 3). The optional "atomic" modifier introduces procedures with backtracking capabilities. "atomic" means that this procedure may be executed in total, or it may be canceled and all changes made by this procedure are revoked.

By default, all variables are defined in a global scope. Every subroutine can see and modify all variables used in the main program. Also, it may set new variables that have not been used in the main routine so far, and these variables will persist after the procedure has returned. A procedure might also accidently overwrite variables used in the main program if the programmer is not fully aware which variables are in use in the main program at the time the procedure is called. To circumvent this problem, and also to save some memory, "local variables" may be declared. These variables are only visible inside the current subrountine, and are discarded once the procedure returns. The use of local variables is mandatory for recursivly called procedures. Otherwise, consecutive calls would interfere with each other due to globally used variables.

Execution of a procedure ceases if its end is reached (implicit return) or due to an explicit return statement. Return statements may optionally pass a return value. If not, a `null` value is returned instead.

Procedure calls are initiated by a colon, followed by the procedure name and the declared amount of parameter: `:`*`tryCargo`* `(` *`$cargo`* `);`.

3.4 Enhanced for-each

A common application for a for-each loop is to try some statements with any combination of multiple variables denoted by list expressions. The goal is to find any suitable combination of variable configurations. Fasel supports this idea by means of an `forany ...`

```
proc atomic tryCargo ( $C )
local $RES ;
...
endproc;
```

Figure 3: Procedure declaration

`try` and `forall ... try` statement. Both `forany` and `forall` statements take a list of variables and list expressions, just like a regular for-each loop. As a loop body, only a single call to a Fasel procedure is allowed. If the procedure returns `true`, this causes a `forany` loop to exit. A `forall` loop will continue until the procedure returns `false` or no other variable configuration is available.

The ferryman algorithm uses a `forany` loop to find any valid cargo to transfer: `return forany ($C in $Cargolist ;) try :tryCargo($C);` . This searches for a configuration of `$C` in `$Cargolist` such that `tryCargo($C)` returns `true`. The loop is part of the solve procedure's return statement.

As mentioned above, it is possible to include several variables in a `forany` or `forall` loop. The example in figure 4 tests whether a pack of cards is complete. All combinations of color and card are passed to `findCard` to check if the card is present. If it is not found, the loop will exit and return `false`. In this case, `$Color` and `$Card` hold the latest configuration `findCard` was called with and hence denote the missing card. Currently, it is not possible to continue a search in order to find other missing cards. If the pack of cards is complete, the loop returns `true`.

```
$complete := forall (
$Color in [ Diamd,Heart,Spade,Club ];
$Card in [ 7,8,9, ... ];
) try  :findCard  ( $Color, $Card );
```

Figure 4: ForAll loop with multiple variables

3.5 Backtracking

Fasel allows to revoke changes made to the graph by means of "atomic" procedures. In contrast to a regular, non-atomic procedure, these may also be exited using the `revoke` statement. In this case, all changes made to any Fasel variable or to the data model itself by the procedure are revoked and the state prior to the procedure call is restored. The `revoke` statement may also pass a return value.

Currently, revoking a procedure is done by serializing and storing the complete data model when an atomic procedure is entered. A `revoke` statement loads the stored model and replaces the current one, whilst a regular `return` statement discards the remembered information and hence makes all changes persistent. Obviously, this approach is not well-suited for large graphs or a large amount of backtracking information to be remembered. Memory demand scales linearly with the backtracking depth which correlates to the amount of recursive calls to atomic procedures. Our example is restricted by the fixed ceiling of seven steps before backtracking is enforced. Also, the graph is limited to a fixed amount of seven nodes which allows solving the problem in less than half a second on a modern PC. More complex tasks clearly demand a more efficient and less memory consuming approach. We will discuss an alternative in section 4.

Often, graph changes have to be backtracked if a certain test is evaluated to `false`, while changes have to be made persistent if it is evaluated to `true`. This is done by the " `decide expr;`" statement. If the given expression is true, this statement is equivalent to `return true;`, otherwise it is equivalent to `revoke false;`.

3.6 Restrictions

Fasel does not consider any Programming-in-the-large aspects like modularization. Also, no type concepts were considerated and there is no need to declare a variable, except for local use in subroutines. Fasel was designed for small scripts that do not require a more sophisticated structure. Fasel scripts should be used to implement algorithms in a recursive or iterative form where Fujaba Story Diagrams are not very well suited. More complex operations like tests for graph patterns should be realized using Fujaba Story Diagrams.

4. CONCLUSION

We demonstrated the usefulness of a new scripting language for Java code named Fasel. Other languages are commonly used for running scripts on Java code and often supported by a large community (e.g. Python for Java, [3]). However, Fasel includes a backtracking mechanism that may be used to revoke prior statements. As Fujaba is lacking an automated backtracking mechanism, we implemented the Ferryman Problem which depends on backtracking as a Fasel script.

We are currently working on integrating CoObRA [5] into Fasel and the Ferryman application. CoObRA is an architecture to keep track of changes made to an object. Revoking changes made to the graph model can simply be done by discarding the latest changes, instead of remembering the complete graph in a serialized from. We expect the CoObRA approach to be a lot less memory consuming and better suited for large graphs.

Furthermore, backtracking is a desireable extension for Fujaba. For example, it is useful to be able to revoke a story pattern if a postcondition of some kind fails. Without backtracking, all changes made to the graph model are persistant and cannot be taken back without user interaction. The concept of "atomic" procedures could be extended to "atomic" activities that are able to revoke their changes if an assertion fails. The algorithm presented in section 2 could then be realized as an activity in the Fujaba model. We are investigating how this extension can be implemented as a Fujaba Plugin.

5. REFERENCES

[1] L. Geiger and A. Zündorf. Graph based debugging with Fujaba. Technical report, AG Softwaretechnik Technische Universität Braunschweig, 2002.

[2] Java documentation. http://java.sun.com/j2se/1.4.2/docs/index.html.

[3] Jython. http://www.jython.org/.

[4] U. Norbisrath, P. Salumaa, and E. Schultchen. Fujaba based tool development and generic activity mapping. Technical report, Department of Computer Science III, RWTH Aachen University, 2004. to appear.

[5] C. Schneider. CASE tool Unterstützung für die delta-basierte Replikation und Versionierung komplexer Objektstrukturen. Master's thesis, Carolo Wilhelmina zu Braunschweig, 2003.

Yet Another Association Implementation

Thomas Maier, Albert Zündorf

Software Engineering Group, University of Kassel,
Wilhelmshöher Allee 73, 34121 Kassel, Germany

Thomas.Maier@uni-kassel.de, Albert.Zuendorf@uni-kassel.de
http://www.se.eecs.uni-kassel.de

Abstract
Fujaba already provides sophisticated code generation concepts for UML associations. However, recent bug fixing work on associations has revealed that the Fujaba code generation mechanisms for associations have a maintenance problem. In addition, the excessive number of access methods generated for to-many association roles consume not only large amounts of memory space in the Fujaba tool but they also clutter the generated code. Thus, this paper revisits the code generation concepts for associations.

Key Words: *UML, Associations*

1 Introduction

Code generation for UML class diagrams is well studied and provided by most modern CASE tools. However, code generation for associations still lacks mature implementation strategies in many current tools. This paper revisits Fujaba's code generation strategy for associations and proposes a new approach trading runtime space and time requirements for better tool maintenance and better readability of the generated code.

In principle, UML associations specify bi-directional relationships between objects of two classes. Each association end, a so-called role, may carry its own name and cardinality. Adornments like aggregation, composition or qualification may be used. In addition, to-many associations may be constrained to be ordered or sorted. This variety of association properties is responsible for the complexity of proper association implementations. Figure 1 shows a simple class diagram used as running example for this paper. It represents parts of the board game Mississippi Queen, where players move steamers down the Mississippi river. The river is divided into hex fields.

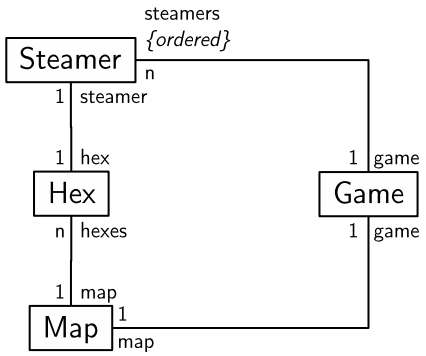

Figure 1: Class diagram Mississippi Queen

In addition we have a number of requirements on a sophisticated association implementation strategy:

- Generally, we expect bi-directional navigability for associations.
- We require automatic integrity between forward and backward navigation. This means, at any time object a refers to object b , object b must have a reverse reference to object a. On creation and deletion of links between objects, both directions have to be maintained, automatically.
- The usage of the association for the application programmer should be comfortable.

- The generation of the associations should be comfortable for the tool developer to ease maintenance.

2 Related work

Most CASE tools provide code generation for class diagrams and associations, these days. However, Together [Bor04] still does not support bi-directional associations and thus does not care about forward backward consistency. Rational [IBM04] allows bidirectional associations but the optional access methods do not guarantee mutual update of forward and backward references. Rhapsody [ILo04] and Fujaba [U+04] provide access methods that update forward and backward references mutually. However, for to-many associations a large number of accessor methods is generated which is not handy, neither for the application nor for the tool developer.

The Meta Object Facility (MOF) approach proposes to use explicit association classes, cf. e.g. [ASB04]. For example an association class Hexes might provide operations like `Hexes.addToHexes (map1, newHex)`. A read access might look like `Hexes.iteratorOfHexes (map1)`. This approach allows to guarantee mutual consistency of forward and backward references. In addition, the explicit association classes facilitate to deal with more complex association features like e.g. the *redefines* or *subset/union* relations between different associations as introduced in UML 2.0. However, still a lot of specific code has to be generated for each association.

In contrast, we use a fixed set of predefined classes. Instances thereof are directly accessible from the linked objects.

Neither approach is arguably "better". It seems to boil down to the developer's mental model whether associations should be first class citizens or not – in other words, it is probably a matter of taste.

3 Associations with Role Objects

The association facilities that Fujaba provides are quite pleasing. Especially the "bidirectional consistency", i. e. the automatic creation and deletion of *both* links between two objects is crucial. However, Fujaba generates two methods for each to-one-role and several methods for each to-many-role a class participates in. This is tedious and error prone for the CASE tool developer and these automatically generated methods clutter the code, unacceptably.

To avoid this, this paper introduces an implementation with explicit runtime role objects to implement bidirectional associations. There is one role attribute for each role (i. e. association end). The role attributes encapsulate all the methods for managing the links between two objects.

3.1 Example Class Diagram

The augmented Mississippi Queen class diagram including our role attributes (still ignoring methods and attributes for the "real" functionality) looks like Figure 2. Note that the role attributes are shown as attributes and not as references to improve legibility.

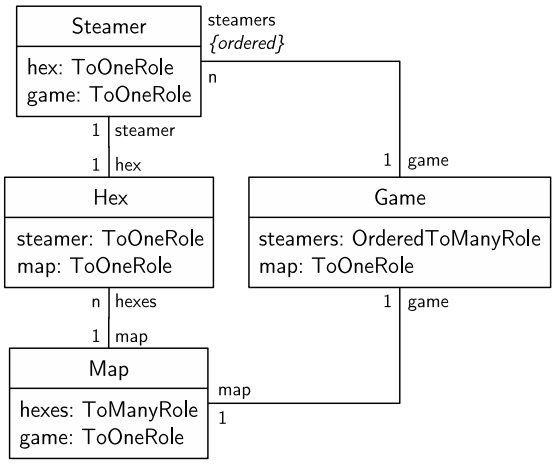

Figure 2: Class diagram with role objects

Basically, ToOneRole attributes are used for to-one roles, ToManyRole attributes for to-many-roles and so on.

In the following a closer look is taken at the one-to-many association between Map and Hex. This association models that a map contains many hex fields and each hex field belongs to a map.

3.2 Example Object Diagram

For each association end (role) a corresponding role attribute is used. This role attribute is named like the role name. At runtime, each

Hex object hex has a ToOneRole object named map to reference the hex's map. Likewise, each Map object map has a ToManyRole object named hexes to reference the map's hexes.

A sample ("logical") object structure is shown in Figure 3 as a UML object diagram. The diagram shows the map object aMap containing the two hex objects hex1 and hex2. Each (bidirectional) link between two objects represents two references, one back and one forth, plus the role objects.

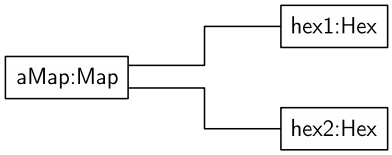

Figure 3: Object structure without role objects

Figure 4 shows the ("implementation") object structure including the role objects that are used for linking the objects. The diagram shows that the objects do not reference each other directly (e. g. aMap does not reference hex1 directly). Instead, linking the objects is done by the role objects. The map object aMap references both hex objects hex1 and hex2 in its ToManyRole object `aMap.hexes()`, or hexesRole (actually the role object uses the collection objectsSet to store and thus to reference the objects). Correspondingly, hex1 and hex2 reference their map object aMap in their ToOneRole objects `hex1.map()` (or mapRole1) and `hex2.map()` (or mapRole2), respectively. So the links are actually bidirectional, although this is harder to see than in Figure 3.

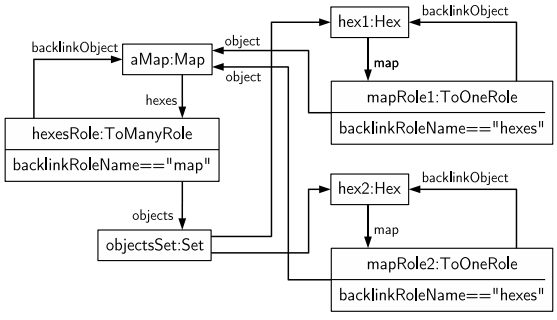

Figure 4: Object structure with role objects

3.3 Usage

Links between objects are modified and queried by invoking operations on the corresponding role objects. Setting the role objects to another value does not make any sense, usually. So only "getter" methods are used for role objects. Setter methods are not used. This is in quotes because those methods are named like the role name, without prepending `get` (altough, of course, it is up to the developer to pick suitable names). Even for a to-one-role roleName the usual getter/setter pair `getRoleName()` and `setRoleName(`*args*`)` is not provided. Instead there is only the method `roleName()`. This leads to the uniform interface for to-one and to-many roles:

```
object.roleName().operation(args);
```

In Figure 5 an example usage with promises written as JUnit [GB04] assertions is shown. The first lines set up an object structure like the one shown in Figure 4, the remaining lines remove the hex objects from the map.

```
Map aMap = new Map();

Hex hex1 = new Hex();
aMap.hexes().add(hex1);

Hex hex2 = new Hex();
hex2.map().set(aMap);

assertSame(aMap, hex1.map().get());
assertSame(aMap, hex2.map().get());
assert(aMap.hexes().contains(hex1));
assert(aMap.hexes().contains(hex2));
assertEquals(2, aMap.hexes().size();

aMap.hexes().remove(hex2);
assert(hex2.map().get() == null);
assert(!aMap().hexes().contains(hex2));

hex1.map().unlink();
assert(hex1.map().get() == null);
assert(!aMap.hexes().contains(hex1));

assertEquals(0, aMap.hexes().size();
```

Figure 5: Example usage

When hex1 is added to aMap by saying `aMap.hexes().add(hex1)`, the link from aMap to hex1 is established. hex1 is not only stored in the role object hexesRole's collection hexesSet. hexesRole also (automatically) calls `hex1.map().set(aMap)` to establish the

so-called *backlink* from hex1 to aMap. The backlink is the link that is automatically created (or removed) to ensure proper bidirectionality.

It works just the other way round when saying `hex2.map().set(aMap)`. This sets the link from hex2 to aMap. The backlink from aMap to hex2 is automatically set by the role object mapRole2.

Deleting a link also automatically deletes the backlink.

To have the role objects do this job, they have to be provided with enough information.

3.4 Role Definitions

The classes participating in an association have to correctly define the role objects implementing the association. Each role object has to know the role name for the backlink and the object that the backlink should reference.

So class Map could implement its end of the Map–Hex association as shown in Figure 6.

```
class Map {
  private ToManyRole hexes;
  public ToManyRole hexes() {
    if (hexes==null) {
      hexes =
        new ToManyRole("map", this);
    }
    return hexes;
  }
}
```

Figure 6: Definition of class Map

Each Hex object hex that gets inserted into a Map's set of hexes by a statement like `aMap.hexes().add(hex)` will automatically get its map object (first constructor argument `"map"`) set to the map object it gets added to (second constructor argument `this`, which is aMap here). The automatic call that occurs to set the backlink is `hex.map().set(aMap)`.

Of course, the getters might lazily create role objects like shown above or the role objects might be initialized statically or in the constructor. Lazily creating a role object `role` has the benefit of saving memory space when an object is not linked to any other object in that role (e. g. imagine leaf nodes in a tree structure). It comes at the cost of an additional `if (role==null)` for every access to `role`. Initializing a role object statically or in the constructor is the reverse: a slightly faster access at the cost of guaranteed creation of every role object. It is up to the application developer to make the right choice depending on the specific application needs.

The definition of Hex could look like in Figure 7.

```
class Hex {
  private ToOneRole map;
  public ToOneRole map() {
    if (map==null) {
      map =
        new ToOneRole("hexes", this);
    }
    return map;
  }
}
```

Figure 7: Definition of class Hex

Here, each Map object aMap that gets set as a Hex's map by a statement like `hex.map().set(aMap)` will automatically insert the hex object this is called on (second constructor argument `this`, which is hex here) into its set of hexes (first constructor argument `"hexes"`). The automatic call that occurs to set the backlink is `aMap.hexes().add(hex)`.

3.5 Implementation

All the role classes ToOneRole, ToManyRole, OrderedToManyRole etc. have methods to set and delete bidirectional links. They expect their corresponding counterparts (the role objects of the class at the other end of the association) to be properly set up as demonstrated in section 3.4. For ToOneRoles, there is the interface

- `Object get();`
- `void link(Object object);`
- `void unlink();`
- `void set(Object object);`

to get the linked object and to link and unlink (remove the links between) two objects, respectively. `set()` and `link()` are the same methods.

For ToManyRoles the standard Java Collection interface [J2S] is implemented to maximize usage flexibility and to be familiar to programmers from the start. This means there are methods like

- `Iterator iterator();`
- `boolean add(Object object);`
- `boolean link(Object object);`
- `boolean remove(Object object);`
- `boolean contains(Object object);`
- `int size();`
- etc.

Here, `add()` and `link()` have the same meaning.

The basic implementation pattern for the methods setting and deleting links is:

- Set or delete the forward link.
- Use Java Reflection to get the role object at the other side of the association.
- Set or delete the backward link.
- Fire property change events if the object structure has changed.

Of course there are checks to prevent endless recursion and to only set the necessary links once.

Figure 8 shows simplified pseudo code that illustrates that pattern for the `add()` method in ToManyRole. Linking `null` is not allowed and is always checked first. The next line sets the forward link by storing `obj` in `objects` or does nothing at all if `obj` is already contained in `objects`. If `obj` was really added to `objects`, `add()` needs to also set the backlink and fire property change events. `getBackLinkRole()` uses Java Reflection to get the role object at the other side of the association (its name was given as a constructor argument, cf. section 3.4). `setBackLink()` also uses Java Reflection to invoke the `link()` operation on that role object to set the backlink.

3.6 Advanced Usage

The plain to-many roles have "set semantics". When creating a ToManyRole object the way that is shown in section 3.4, they use a default Set object (currently HashSet).

Ordered to-many roles have "list semantics". When creating an OrderedToManyRole object, they use a default List object (currently LinkedList).

```
public boolean add(Object o) {
  if(o == null) {
    throw
      new IllegalArgumentException();
  }
  boolean changed = objects.add(o);
  if (changed) {
    setBackLink(getBackLinkRole(o));
    firePropertyChange(/* added o */);
  }
  return changed;
}
```

Figure 8: Simplified implementation of ToManyRole's `add()` method

There are no seperate role classes for the *sorted* role attribute. Instead, sorting the elements is delegated to the collection. When creating the role object there are two possibilities. Either a default sorted role may be created. A default sorted collection is used, then (e. g. SortedSet for ToManyRoles). Or, alternatively, a specific collection object to use might be passed as an argument. This can be useful when using a custom set of collections, e. g. to work around the severe limitations of the standard Java Collection iterators (although this has been partly addressed by the new J2SE version 5). It is also useful for setting up a SortedSet with a custom Comparator.

4 Conclusion and Future Work

The association implementation introduced here implements bidirectional consistency. They fulfil our functional needs. The interface is designed to be as uniform and as familiar as possible. After using it for a couple of months the authors think it is very handy and easy to use for the programmer. The same should apply for the tool developer. Both will be tested while developing the Janus Plugin [M+04].

Currently, we perform performance tests in order to compare time and space efficiency of the new approach with Fujaba's conventional association implementation.

In future versions of the package the long awaited generic type feature that will be available with Java version 5 will be used. A package being close to collections like this one is predes-

tined to exploit the improved type safety and convenience. The interface has actually been designed with generics in mind.

In our experience it is not really important to have specific cardinalities like 7..42. The major difference seems to be to have either *one* object or *many* objects at the other end of the association. Other people probably think differently about this and so this feature might be incorporated in the future. It could also be useful when designing software for embedded systems. Usually, those systems lack dynamic memory management. So using arrays as the collections to store objects is natural there. However, arrays always have at least an upper bound so being able to specify an upper cardinality is a feature that is likely to be implemented in the future.

Qualified associations are hard to get "right". It is difficult to meet everybody's needs here. There is basic support for qualified associations (qualifying by an external key and qualifying by an object's attribute). Extending this functionality is planned.

UML 2.0 introduces the attributes *redefine*, *subset* and *union* for roles. [ASB04] is able to deal with these extensions due to the explicit association classes. We believe that the same holds for our explicit role classes. During a brain storming session, we developed some simple mechanisms how this may be achieved. However, these mechanisms still need to be evaluated.

Currently, implementations for most of the required role classes are available at [Mai04]. We have used these mechanism within the Janus project successfully, although the Janus project did not yet use a code generator at all. Current work is the adaption of Fujaba's code generation mechanisms to generate our implementation of associations. However, this also requires an adaption of the code generation for story diagrams and an adaption of the dynamic object browser Dobs.

References

[ASB04] Amelunxen, Carsten, Andy Schürr und Lutz Bichler: *Codegenerierung für Assoziationen in MOF 2.0.* In: *Proceedings zur Modellierung 2004*, Seiten 149–168, Marburg, März 2004.

[Bor04] Borland: *Together CASE Tool*, 2004. `http://www.borland.com/together`.

[GB04] Gamma, Erich and Kent Beck: *JUnit*, 2004. `http://www.junit.org`.

[IBM04] IBM: *Rational Rose CASE Tool*, 2004. `http://www.ibm.com/rational`.

[ILo04] ILogix: *Rhapsody CASE Tool* , 2004. `http://www.ilogix.com`.

[J2S] *Java 2 Platform, Standard Edition (J2SE)*. Sun Microsystems, Inc. `http://java.sun.com/j2se`.

[M+04] Maier, Thomas et al.: *Janus Plugin – Java'n'UML Simultanously*, 2004. `http://janus-plugin.sourceforge.net`.

[Mai04] Maier, Thomas: *Associations*, 2004. `http://associations.sourceforge.net`.

[U+04] Universities of Paderborn, Kassel, Darmstadt et al.: *Fujaba – From UML to Java and Back Again*, 2004. `http://www.fujaba.org`.