Holger Giese, Albert Zündorf (Eds.)

FUJABA Days 2005

15th -18th September 2005
Paderborn, 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. 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

Holger Giese (University of Paderborn, Germany)
Albert Zündorf (University of Kassel, Germany)

Program Commitee Members

Gregor Engels (University of Paderborn, Germany)
Pieter van Gorp (University of Antwerp, Belgium)
Sabine Glesner (University of Karlsruhe, Germany)
Luuk Groenewegen (Leiden University, Netherlands)
Reiko Heckel (University of Leicester, UK)
Jens Jahnke (University of Victoria, Canada)
Mark Minas (University of the Federal Armed Forces, Germany)
Manfred Nagl (RWTH Aachen, Germany)
Jörg Niere (University of Siegen, Germany)
Bernhard Rumpe (TU Braunschweig, Germany)
Andy Schürr (TU Darmstadt, Germany)
Wilhelm Schäfer (University of Paderborn, Germany)
Dániel Varró (Budapest University of Technology and Economics, Hungary)
Bernhard Westfechtel (University of Bayreuth, Germany)

Editors' preface

Fujaba is an Open Source UML CASE tool project started at the software engineering group of Paderborn University in 1997. It initially combined features of commercial "Executable UML" CASE tools with rule-based visual programming concepts adopted from its ancestor, the graph transformation tool PROGRES. 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.

Fujaba followed the model-driven development philosophy right from its beginning in 1997. 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, Darmstadt, and Siegen 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.

Quite a number of research groups have also chosen Fujaba as a platform for UML and MDA related research activities. In addition, quite a number of Fujaba users send us requests for more functionality and extensions.

This 3rd International Fujaba Days aims at bringing 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.

Thanks to the EU research project SEGRAVIS we were able to invite Jean Bezivin to give a keynote talk on ModelWare for the Fujaba Days. Within the technical program of the main workshop, we will have 10 papers addressing meta-modeling, reverse engineering, formal methods, and code generation and real-time systems. An additional demo session and two panel sessions will complete the program.

The abstracts of the invited talk plus an overview paper about the development efforts at the main Fujaba locations Paderborn, Kassel, Darmstadt, Siegen, and Bayreuth, 5 position papers presenting innovative ways of using or extending Fujaba of about 8 pages, and 4 reports of new features or projects extending Fujaba of about 4 pages 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 will be one session on model-driven development, one session on reverse engineering, one session on formal methods, one session on code generation as well as one session dealing with real-time systems.

We hope that this compilation of the keynote abstract, the overview paper, the position papers, and the reports of the Fujaba workshop presentations provides a useful insight into the ongoing activities in the Fujaba project and provides the needed background information and motivation for joining the Fujaba development team.

The Fujaba Developer Days after the main workshop further enable active Fujaba developers to meet each other and to spend two days together to share their experiences, to work on Fujaba extensions, etc. Within the Fujaba Developer Days, a number of tutorials about Fujaba 4 Eclipse, the new code generation mechanism, changes due to the upcoming Fujaba 5.0 release, and the Support for different meta-models will be held.

Holger Giese & Albert Zündorf
Program Committee Chairs

Table of Contents

The Fujaba Tool Suite 2005: An Overview About the Development Efforts in Paderborn, Kassel, Darmstadt, Siegen, and Bayreuth

The Fujaba Developer Teams from
Paderborn, Kassel, Darmstadt, Siegen, and Bayreuth

ABSTRACT

This paper provides an overview about the Fujaba Tool Suite and its independent tool versions developed and maintained in Paderborn, Kassel, Darmstadt, and Siegen including future plans for tool development in Bayreuth. Starting with the architecture and roadmap for the Fujaba core, we report about the different directions of development (e.g. meta-modeling, realtime systems, and re-engineering) and also provide an outlook on planned future work for each direction. In addition, references which provide more detailed information for the different directions are provided and a comprehensive list of the plug-ins which are available or currently under development by Paderborn, Kassel, Darmstadt, or Siegen is included.

Keywords

Fujaba, UML, MOF, JMI, Reverse Engineering, Design Pattern, Anti Pattern, Metrics, Model-Driven Development, Code Generation, Story Pattern, Graph Transformations, Triple Graph Grammars, Tool Integration, Versioning, Real-Time, Mechatronic, Modelchecking, Scenarios, Teaching, Eclipse, Metamodeling

1. INTRODUCTION

Fujaba is an Open Source UML CASE tool project started by the software engineering group of Paderborn University in 1997[1]. 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.

Fujaba followed the model driven development philosophy right from its beginning in 1997. 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 suites are under development in Paderborn, Kassel, Darmstadt, and Siegen for supporting (1) re-engineering, (2) real-time and mechatronic 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.

The paper is organized as follows. In the next section, we report about architectural improvements concerning the Fujaba kernel and about the Eclipse integration efforts as well as approaches concerning model versioning. In Section 3, we present our approaches concerning model-driven development. This includes work on support for MOF 2.0 meta-models as well as various applications of triple graph grammars for e.g. tool integration and (meta-)model transformations. We show in Section 4 ongoing research concerning reverse engineering of software systems, in particular design pattern and anti pattern recognition. Section 5 contains a presentation of the various approaches concerning model driven development of dependable, embedded, real-time systems. In Section 6, we present work on teaching object-oriented concepts using Fujaba as well as work on a Fujaba supported software development process. We conclude the paper with an overview about possible new research areas for Fujaba. [hg,mtt]

[1]Fujaba is a successor of the not UML-compatible CASE tool PROGRES whose development has been started around 1986.

2. MODEL MANAGEMENT - ARCHITECTURE AND ALGORITHMS

In this section, we report about the current state of the Fujaba kernel as well as the current state of the Fujaba Eclipse integration efforts. This section includes reports about versioning in Fujaba and work on computing differences between models.

2.1 Kernel

The FUJABA TOOL SUITE has been well proven in various projects in the last eight years since its beginning. Especially, the changeover from FUJABA 3 to the FUJABA TOOL SUITE 4 with its ability to add plug-ins had opened our project to a wider developer community beyond the University of Paderborn. However, the FUJABA TOOL SUITE has still limitations that impede further development.

FUJABA can only handle one project at a time. This prevents the possibility to define dependencies between projects. Projects must always be consistent in itself. Future versions of the FUJABA TOOL SUITE will be able to handle more than one project with project dependencies.

FUJABA is particularly well known for its *Story Driven Modeling* (SDM). However, SDM is restricted to the UML class meta model. There are new approaches using the *Meta-Object Facility* (MOF) for class models. To support such approaches, an interface for class meta models and SDM is introduced so that the meta models can be changed [46]. Further enhancements of the meta model and the user interface are under work.

All these enhancements will result in the FUJABA TOOL SUITE 5 which are in turn only a preparation for the most

important enhancement. The FUJABA TOOL SUITE will be migrated to the widely used IDE ECLIPSE. [lw,tr]

2.2 Fujaba4Eclipse

In 2004, we received an IBM Eclipse Innovation Grant for porting a core part of the Fujaba Tool Suite to an Eclipse plug-in [35]. Today, the third release of FUJABA for Eclipse is available for download at the FUJABA web site. So far, the plug-in supports modeling of class and story diagrams, from which Java code can be generated.

The Fujaba4Eclipse plug-in reuses major parts of FUJABA. The display and editing of diagrams has been reimplemented using the well-documented Eclipse and GEF frameworks. First tests show that especially the display of large class diagrams is much faster than in FUJABA.

Fujaba4Eclipse already provides several extension points which enable the contribution of new diagrams or diagram elements as well as loading and saving of model elements provided by other Eclipse plug-ins. More extension points will follow. Several FUJABA reverse engineering plug-ins have already been ported successfully to Eclipse. The experience shows that the effort required is not too high since many parts of Fujaba4Eclipse, e.g. figures for displaying UML elements, can easily be reused. The MyPlugin example plug-in for Eclipse will soon be available as well, so start contributing! [mm,lw]

2.3 Persistency [KS]

The old storing mechanism of Fujaba (FPR) accompanied the Tool right from the first development stages. But as Fujaba evolved, more and larger projects were started, using Fujaba. Thus a single-user environment is not suited for these projects any more. Previous attempts to add a versioning facility to Fujaba [45] did not succeed.

CoObRA is a framework, developed at the University of Kassel, that offers undo/redo, persistency and version control techniques [15]. It records changes to the object structure of an application to facilitate all these features. The major advantage is that it provides an easy-to-use mechanism requiring very low integration costs. CoObRA was successfully integrated into the model of the Fujaba Tool Suite.

This results in a major feature for Fujaba: Projects can be stored in local files and in server side repositories to share them among developers, both with the same API. Additionally full undo and redo support was added to Fujaba. For applications developed with Fujaba the CoObRA plug-in also extends the Fujaba code generation facilities to offer CoObRAs features to applications generated with Fujaba.

CoObRA2, which is integrated into Fujaba5, already provides all the features from CoObRA and was optimized concerning design and processing speed. Planned features include hierarchical and cascaded repositories to support working in groups at different development sites. Also, it is future work to use CoObRA2 with CVS servers, to concurrently work on e.g. Fujaba project files and additional resources like documentation and images within the same CVS repository.

To continue the work on Fujaba's scalability for large projects, we want to introduce a generic loading on demand strategy. Models could be partially loaded when required and unloaded to free memory without harming proper operation. This would tackle the memory problems with large Fujaba specifications, which are quite some obstacle at this time. [cs]

2.4 Differences of Models

In general, software should be developed in teams. In order to support cooperative team work a version management system which supports UML models is absolutely necessary. It is not always possible to record changes to a model as done in the CoObRA approach, depicted above. Thus the essential part of such systems is the ability to calculate differences, present them to the developer and to provide merge operations to come to a consistent model. In our approach we use unified diagrams with difference information to be presented to the developer.

The current algorithm calculates differences of two models given as XMI files. The output is also an XMI file; it contains the unified model of the two original models with additional difference information added by the XMI extension mechanism. The difference algorithm does not rely on persistent unique identifiers and the calculation itself is configurable to capture the semantics of an actual model or part of the model in the algorithm. The algorithm is designed to support flat diagrams, where a recursive nesting of diagram elements is forbidden, e.g. UML class diagrams or sequence diagrams. The configuration of the algorithm for UML class diagrams works well and is part of the plug-in distribution. Currently the algorithm is enhanced to support hierarchical diagrams such as UML package diagrams, state-charts or Matlab/Simulink models. For more details see [27].

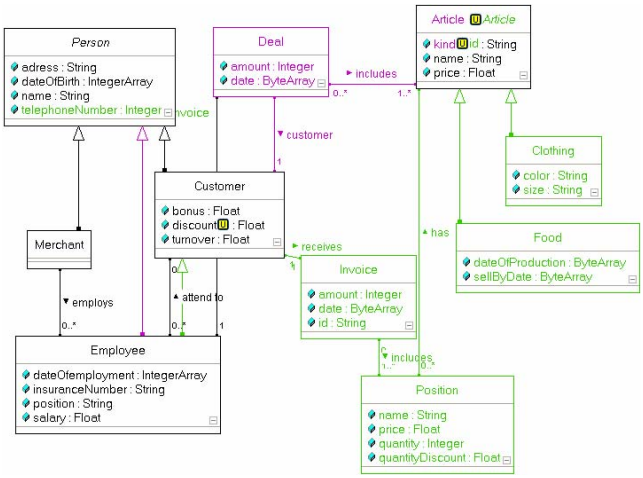

Figure 1: Document with difference information. Parts detected as similar are shown in black color, the green (light gray) parts exist only in the second diagram and the red (dark gray) parts exist only in the first diagram. The small u-button indicates a value change.

Unfortunately, a difference diagram type has less restrictions than a 'normal' diagram type. For example a difference class diagram has to allow for having more than one class with the same name or a class must be allowed to contain two methods with the same signature having only different parameter names. Consequently, each difference diagram type needs its own visualization. A configurable algorithm such

as the calculation algorithm is not feasible. We have developed a cook book which describes how a diagram type implementation can be transformed in an implementation for the corresponding difference diagram type step by step. Fujaba currently supports the visualization of difference class diagrams only, cf. Figure 1. For more details see [37, 41]. [jn]

3. MODEL- AND META-MODEL-DRIVEN SOFTWARE DEVELOPMENT

Originally, Fujaba has been developed to support model-driven software development activities by offering its users an executable subset of UML 1.x with a precise semantics definition based on graph transformations. Despite of the fact that Fujaba has been used from the very beginning for meta modeling and bootstrapping activities, the tool's main focus was and still is the rule-based development of ordinary Java applications with complex data structures. OMG-compliant meta modeling activities and the definition of model-to-model transformations were out-of-scope. Nowadays this situation is rapidly changing with the development of a number meta modeling and model-transformation plug-ins, which are the main topic of this section.

These plug-ins support meta modeling activities based on MOF 2.0, rule-based definition of "local" model transformations (working on a single model on the same abstraction level) as well as the specification of step-wise model refinement processes and the translation between models belonging to different meta models. These plug-ins developed at different Universities, are described in the following subsections. [as]

3.1 Meta Modeling with MOF 2.0

Meta -modeling languages are used to describe other modeling languages. The Object Management Group (OMG) adopted MOF 2.0 [40] as standard meta modeling language. It provides valuable new features for modularization, refinement, and reuse of meta models compared to UML 1.x class diagrams which are currently supported by the Fujaba kernel.

Therefore, we developed a meta modeling framework called MOFLON (www.moflon.org). The framework is based on MOF 2.0 and realized as a new Fujaba plug-in [2] (replacing the Fujaba UML 1.4 compatible class diagram editor). MOF 2.0 simplifies modularization and reuse of meta models by offering a sophisticated package concept as well as powerful means for refinement of attribute types and associations. Packages act as namespaces and can be refined and reused using package merges. This concept is more sophisticated than Fujaba's current package concept.

Further major features of MOF 2.0 compared to Fujaba UML class diagrams are the relations between association ends. In MOF 2.0 association ends can be redefinitions, subsets or unions of other association ends. Those relations simplify the challenging task to structure, reuse, and refine huge meta models. Beside the new features, the editor offers an XMI import which permits the exchange of models with Rational Rose and other XMI-standard-compliant commercial UML CASE tools. [ca,tr]

3.2 Meta Model Transformation with Story Diagrams

Choosing MOF 2.0 as modeling (graph schema definition) language has some implications for Fujaba graph transformations as discussed in [3]. Model/graph transformations defined by means of story diagrams, are no longer manipulating simple directed, node- and edge-labeled graphs with attributed nodes, but have to be adapted to the much more complicated data model imposed by MOF 2.0. This adaption involves some modifications of different parts of Fujaba. On the one hand, packages offer now much more sophisticated means to structure, reuse, and refine model fragments and to define namespaces for classes and other model elements; they are intended to replace Fujaba's simple view diagram concept. On the other hand some effort has be done to integrate the improved association concept that is introduced by MOF 2.0 as well as UML 2.0 into Fujaba graph transformations, i.e. to modify its underlying class of graphs appropriately.

In MOF 2.0, set-valued association ends are not necessarily unique. As already indicated in [53], the introduction of non-unique association ends requires modifications of the graph model, because story diagrams have to deal with multiple links of the same type between identical objects. Besides, relationships between association ends can be defined that result in automatic link-creation (subsets) or restrict possible end types for given associations(redefinition). While automated link-creation is transparently performed by the MOFLON plug-ins [2], some modifications are necessary in the Fujaba core to support the remaining new MOF 2.0 concepts: Unions of association ends make associations virtually abstract, so links of that type may be queried, but not created, modified or deleted by story diagrams. These restrictions should be checked statical during the creation of story diagrams. Other new features like redefinition of association ends demand additional runtime analysis activities. [ca,tr]

3.3 MDD and Stratified Graph Transformations

Complex software systems call for ways to obtain simplified views onto them, otherwise important architectural structures will be buried in overwhelming details. However, resolving certain issues often requires the manipulation of more detailed architecture without destroying the consistency between these more detailed models and their abstractions. Architecture Stratification [4] addresses the need to provide a multitude of views on models ranging from very concrete to very abstract so that a large number of stakeholders may view the system (model) with a level of abstraction that is best for their particular interests and requirements. In contrast to a layer, as in the ISO/OSI network model, each level fully describes the whole system, albeit using varying conciseness.

The SPin plug-in [28] extends Fujaba with facilities to annotate very abstract models so that they can be successively transformed into more concrete versions in accordance with OMG's MDA approach. SPin comes with a dynamically extendable library of refinement transformations. Thus, SPin provides basic support for architecture stratification. Currently, refinement transformations are realized as straightforward Fujaba graph transformations. Future work will investigate the use of triple graph grammars (TGGs) for the same and more advanced purposes including round-trip engineering. [fk,tk]

3.4 Model Transformation using TGGs

The advent of OMG's new silver-bullet of the model-driven architecture (MDA) [42] has put model transformations into focus. A practicable model transformation technique should allow a visual specification of the transformation with an underlying formal foundation and is expected both to be applicable in different stages of the development process and in both directions. The transformation process itself should be executable in a batch-oriented and incremental way. Additionally, the technology should allow synchronizing models that belong to different meta models and keeping them consistent.

Due to these requirements, we are using the visual, formal, and bidirectional transformation technique of triple graph grammars (TGGs) [48]. A triple graph grammar specification is a declarative definition of a bidirectional model transformation. The mapping between two models is achieved by an additional correspondence model. It enables a clear distinction between the source and the target and holds additional traceability information [12]. This extra information is needed to preserve the consistency between both models.

A recently developed Fujaba plug-in offers the required tool support for model transformations with triple graph grammars. Its main component is an editor for the declarative specification of triple graph grammar rules. From this specification we derive implementations of executable consistency-checking or establishing rules automatically. The rules are executed using the standard Fujaba model transformation engine. This engine is currently extended towards incremental model transformations. Additionally, we have developed a concept which allows us to transform Fujaba incompliant models. Based on this concept, we have developed a code generator with round-trip engineering capabilities. The generator transforms class diagrams represented in Fujaba to Eclipse's Java abstract syntax tree and vice versa. This case study is a first step towards model round-trip engineering and code generation realizing the vision of MDA. [rw]

3.5 Tool Integration with TGGs

As mentioned in the previous subsection, triple graph grammars provide powerful means to integrate models belonging to different meta models. In contrast to the activities discussed above, where Java tools with open interfaces are integrated for round-trip engineering purposes, the main focus of the work described here is laid on consistency checking and update propagation between models manipulated by commercial-off-the-shelf modeling tools. These tools are neither written in Java nor do they usually offer appropriate interfaces for model manipulation purposes. Furthermore, automatic round-trip engineering is usually unfeasible and the main emphasis has to be laid on consistency checking and traceability link creation activities. In addition, adherence to standards is a must which are or will hopefully be in the future supported by the regarded CASE tools.

Therefore, we are developing a variant of TGGs which finally generates Java code for MOF/JMI-compliant tool interfaces. Furthermore, we take OMG's MOF 2.0 Query / Views / Transformations Request for Proposals (QVT) [43] and the most promising submission from the QVT-partners [44] into account. Our solution is based on the tool integration framework Toolnet from Daimler Chrysler [1]. This framework offers interfaces for uniformly accessing tools' data by means of so called tool adapters based on Java Metadata Interfaces (JMI) [36]. Finally, the framework provides the infrastructure for inter-tool communication. We are currently busy to extend Toolnet by (semi-)automatic tool integration mechanisms (e.g. consistency checking, attribute change propagation, model transformation). In this scenario, Fujaba is used to draw declarative triple graph grammars, (semi-)automatically derive operational graph rewriting rules for the different tool integration use cases, and generate JMI-compliant Java code from them. In the future, we plan to extend the formalism of triple graph grammars to multi graph grammars. With multi graph grammars, we will be able to cope with an arbitrary number of to be integrated tools instead of pairs of tools only [32]. [ak,as]

3.6 Incremental Model Synchronization using TGGs

One common problem application programmers in general and CASE tool developers in particular have to solve is to keep some internal object model and its representation on screen consistent, i. e. have the graphical user interface always display what the internal object model looks like.

Although this is not a "hard" problem and everybody knows how to implement such a synchronization, it is at least annoying and error prone work that keeps you away from the "real" work. Usually observers are registered on both models. They are activated when something changes in one model and then update the other model. So for each change operation an observer reaction has to be implemented.

In our experience, when implementing observers you usually think in terms of "what happens if attribute x changes its value" or "what happens if object y gets another child object of type Z". This is a rather low level operational kind of abstraction close to the implementation of observers. A higher level of abstraction is to think in terms of "what-corresponds-to-what-rules" and leave the actual operations necessary to perform the synchronization to a tool that does the right thing.

We think that triple graph grammars are a good match for that higher level of abstraction. We think that those rules can be written down using triple graph grammars, maybe with some extensions added. We plan that this is done in such a way that either the observer code can be automatically generated or that an interpreter executes those rules. This will result in two major improvements compared to manual implementation. First, you will not have to implement the observers manually any longer. Second, designing and maintaining the synchronization can be done on the higher level of abstraction described above.

That scheme is not limited to the special problem described here which represents a part of our motivation. Because we use triple graph grammars it is straightforward to apply that scheme to any two object models. [tm]

3.7 CodeGen2

The old code generation concept of Fujaba added an abstraction layer that should ease the adaption to new programing languages. But that also makes the maintenance of the code generation very complex. In fact, to generate few line of code one has to write dozens of lines in the code generation. This makes small adaption of the code generation impossible especially for Fujaba users. The CodeGen2 plug-

in now replaces the old code generation by a new template-based one. Using templates, it should be possible (even for the user) to adapt the generated code to ones wishes and to write own templates for new languages or new language elements.

Additionally, CodeGen2 introduces a layer of so called tokens on which the templates are applied. On this layer optimizations can easily be preformed. Several optimizations have already been implemented which results in faster execution time of the generated code.

CodeGen2 was completely specified using Fujaba (except for the templates, of course). This way CodeGen2 is the first step towards bootstrapping of Fujaba. [cs,lg,cr]

3.8 JMI Code Generation for MOF 2.0

One part of the MOFLON framework as described in 3.1 is a code generator plug-in based on the MOMoC compiler [6] for the generation of MOF 2.0 compliant meta models. To be able to work with meta modeled languages, tool providers need to map meta models on source code. For that purpose, SUN provides the Java Metadata Interface (JMI) [17] which is a standardized mapping of MOF compliant meta models onto Java. As the Java representation of MOF, JMI specifies the generation of tailored interfaces for the creation and access of meta data as well as a set of reflective interfaces for a unified discovery of meta data. The standardized interfaces facilitate an easy exchange and integration of meta models which make JMI a very useful standard.

Thus, the MOFLON code generator adheres to the JMI standard and generates compliant meta model implementations. The XML-based code generation process consists of three steps. First of all a modularized preprocessing phase performs several kinds of transformations on the meta model (e.g. unfolding of refinement relationships between packages). Afterwards in the second step, the meta model is transformed into a proprietary XML representation which acts as input for several XSL stylesheets transforming the XML representation during the third and last step into Java code. The generated meta model implements all JMI interfaces and supports the new features of MOF 2.0. Furthermore, all changes concerning the instances of the meta model are propagated by an event mechanism to enable a clean and easy integration. This event mechanism is based on the event mechanism provided by the NetBeans Metadata Repository (MDR) [33], a JMI repository which is used by several tools, to achieve compatibility between MDR and meta models generated by MOFLON. Future versions of the code generator will be integrated with an OCL compiler and the graph transformation of Fujaba (cf. Section 3.2). Therefore, the MOMoC/XSLT-based three-phase code generation approach has to be integrated with the template-based code generation mechanism of CodeGen2 described in the previous subsection. [ca]

4. RE-ENGINEERING

Today, software is seldom developed from scratch. In fact, developers spend most of their time with the maintenance of existing systems, e.g. to meet new requirements or to integrate them with other software components. Unfortunately, (design) documentation for such systems or components is often not available or has become obsolete over time. Thus, before changes can be made, an understanding of the system has to be gathered from the available artifacts such as e.g. the source code.

The work presented in the following sections aims to support this task. The first approach reverse engineers class diagrams from the byte code of Java class libraries in order to make them amenable for model based development. The following approaches are concerned with the analysis of source code in order to detect or improve design and anti pattern implementations as well as to perform coarse-grained analyses to get a first impression of a system. Finally, an approach for analyzing the evolution of domain-specific software architectures is presented.

4.1 Java Byte-code

Fujaba's *Story Driven Modeling* (SDM) is very helpful for creating method implementations. Especially pattern-matching is well supported by SDM, because of the graphical specification approach. However, to be able to use SDM, a class diagram containing the metamodel of the elements to be used in SDM must be present in the current Fujaba project.

If a developer just wants to use his own model elements which are defined in his project, this does not represent a problem. Yet, often developers use elements of other developers. For those elements typically no class diagram exists, but corresponding source code or byte code only. To get a class diagram from these representations one can reverse engineer "by hand" or use a parser in combination with a generator that produces a class diagram automatically. A plugin that is able to automate this process for Java source code is the JavaParser plug-in which is part of the *Fujaba RE Tool Suite*.

An alternative way to obtain the desired class diagrams is the *Java Virtual Machine synchronizer* (VM-synchronizer) introduced in [28]. It is part of the "SPin" plug-in and supports synchronization of UML classes with their Java counterpart by creating class diagrams from Java byte code. [fk]

4.2 Design Pattern Recognition

In recent years, we have developed a semiautomatic design pattern recognition on the source code of existing systems [38]. The approach uses a static analysis of the abstract syntax graph (ASG) representation of the source code. Patterns are described graphically based on graph grammar rules with respect to the ASG. It is a highly scalable process which can be applied to large real world applications with more than 100.000 LOC. The found design pattern instances are rated by accuracy values [39].

The static analysis focuses on the structural aspects of a pattern. Behavioral aspects of patterns can only be analyzed rudimentary by static analysis. Thus, we are working on a dynamic analysis that confirms or weakens the pattern instances from static analysis [51]. Behavioral patterns based on sequence diagrams describe the interaction within patterns. The dynamic analysis executes the system to be analyzed and records traces of method calls. These traces will be compared to the behavioral patterns. [lw]

4.3 Anti Pattern Recognition and Transformation

In contrast to design patterns, anti patterns represent bad design solutions especially with respect to maintainability. Examples include data classes consisting only of fields and

almost no behavior, god classes assuming too much responsibilities in a (sub-)system, or huge conditionals for handling requests or representing different states of an object. The presence of anti pattern instances in a system may complicate certain maintenance tasks significantly. Thus, we use the pattern recognition approach introduced in the previous section in combination with software product metrics to recognize anti pattern instances.

An anti pattern can usually be transformed into a better solution, sometimes based on design patterns. Starting from the ideas presented in [26], we are currently working on support for the improvement of recognized anti pattern instances. We want to provide anti pattern specific refactorings of the ASG, which improve the structure of the software while preserving its behavior. Since not necessarily all found anti pattern instances need to be improved, the re-engineer determines those to be transformed. The transformation itself, however, we plan to automate as far as possible. [mm]

4.4 Metrics

Software product metrics are one opportunity to perform coarse-grained analysis. Metrics such as lines of code, number of attributes or methods of a class, lack of cohesion, or depth of inheritance hierarchies allow for producing quantitative analysis results of a software system. A combination of different metrics allows to draw conclusions such as problematic or high influencing system parts. To overcome the flood of numbers produced by the metrics, Lanza proposes a graphical representation of metric combinations. So-called polymetric views are an ideal means to get a first impression of a system.

Polymetric views are two-dimensional graphs containing nodes and sometimes edges, which are arranged in a certain layout. Nodes represent entities of the analyzed software system and edges represent relationships between the entities. Each node is a rectangle and can carry up to 5 metric values depending on the certain layout, whereas edges do not carry any metric information. A reverse engineer can easily identify runaways by looking at the produced pictures and perform further analysis.

Fujaba needs two plug-ins to support the metrics calculation and polymetric views. The first plug-in offers the calculation of several object-oriented software product metrics. The second plug-in allows for viewing the metric results calculated by the first plug-in as polymetric views. Both plug-ins allow for extending and modifying the set of metrics and polymetric views, respectively. For more details see [34]. [mm,jn]

4.5 Trend Analysis of Domain-Specific Software Architectures

The aim of this research activity is to define a methodical approach to monitor the progress of evolving a domain-specific software architecture from an initial to a desired situation. Therefore, both relevant metrics and consistency rules are taken into account. Rather than gathering architectural information only once to assess the quality, we repeat the measurements over an extended period of time and analyze the trends that can be observed. A case study of this approach has been described in [47].

Fujaba is used as meta modeling tool to define the analysis part of the related tool chain. First, the domain-specific meta model is defined using the MOFLON/Editor plug-in described in section 3.1. Consistency rules and metrics are then defined using Fujaba graph transformations that have been adapted to MOF 2.0 as described in section 3.2, and triple graph grammars (cf. section 3.5). Using the JMI-compliant MOFLON/Compiler plug-in described in section 3.8, Java code for analysis rules is generated, which can be checked and visualized by a partially hand-written tool framework. [tr]

5. REAL-TIME AND MECHATRONIC SYSTEMS

A new field of research concerns the software development for reconfigurable mechatronic systems. Mechatronic systems combine technologies from mechanical and electrical engineering as well as from computer science. They are *real-time systems* because reactions to the environment usually have to be completed within a specific, predictable time and they are *hybrid systems* because they usually consist of discrete control modes as well as implementations of continuous feedback controllers. Due to their application domain, the behavior, which is largely controlled by software, has to meet safety-critical requirements. Thus, modeling approaches for mechatronic systems must include formal verification in order to guarantee safety-critical requirements. In the following sections, we describe our ongoing efforts for modeling and formal verification of mechatronic systems. [mtt]

5.1 Modeling

In order to support specifying the architecture of embedded real-time systems which are usually distributed systems, Fujaba has been extended by UML 2.0 component diagrams in the Fujaba Real-Time Tool Suite [11]. The behavior of single components is specified by *real-time statecharts* [9]. Real-time statecharts introduce clocks and allow the specification of worst-case execution times (WCETs) for activities and deadlines for transitions. The abandonment of the zero-execution time semantics for transitions enables code generation for hard real-time systems. We introduced coordination patterns [24] as a notion to capture the local real-time coordination behavior of different roles as real-time statecharts. Those patterns are subject to formal verification in order to guarantee safety-critical properties. Component behavior is then derived by composing and refining a number of pattern role behaviors. In addition, story diagrams have been enhanced to automatically derive worst-case execution times [13]. Code generation exists for Real-Time Java [7] and for C++. [sb]

5.2 Modelchecking

Mechatronic components, which beside their local control are also interconnected with each other, result in a complex distributed embedded real-time system. As such mechatronic systems often contain real-time and safety-critical requirements, a proper approach for the real-time and safety analysis is mandatory. One aim of the Fujaba Real-Time Tool Suite is to provide a tool support for verification. Therefore, a number of plug-ins were developed. The main plug-in provides compositional model checking techniques and a background model checking engine. An interface permits to attach different model checker backend plug-ins, e.g. Uppaal and Raven, realizing a transformation

from the UML model to the model checker specific input language [24]. [mh]

5.3 Invariant Checking

In addition to the modelchecking of the interaction between components, the Fujaba Real-Time Tool Suite offers a plug-in which is capable of checking story patterns. Properties to be checked are structural ones given as so called forbidden graph patterns. Each forbidden graph pattern describes a critical situation and must never be part of a concrete object diagram produced by the application of a given set of story patterns. The developed plug-in automatically checks for a given set of forbidden graph patterns whether a set of given story patterns can ever produce one of them. If this is possible the plug-in outputs a counter example showing the faulty story pattern and how it can produce an object diagram containing a forbidden graph pattern. In contrast to most other tools on checking graph transformation systems our approach is also feasible for infinite state systems (cf. [23] and [5]). [ds]

5.4 Hybrid Systems

Mechatronic systems, which combine techniques from mechanical and electrical engineering and from computer science, require the integration of components from these different domains. This has been achieved by enhancing the Real-Time Tool Suite with hybrid components [21]. Hybrid components integrate for example feedback-controller components from the control engineering domain with *hybrid reconfiguration charts*[2] which are extensions of real-time statecharts (see Section 5.1).

Hybrid components and hybrid reconfiguration charts do not just provide the required integration of the different domains, they further reduce complexity of the models when specifying dynamic reconfiguration and simplify analyses [21]. We integrated the real-time modelchecking with the design of hybrid systems [14] and support code generation for C++.

Currently, we extend hybrid reconfiguration charts to model reconfiguration by graph-based *reconfiguration rules* [10] and to support flexible resource management [8]. [sb]

5.5 Multi-Agent Systems

Mechatronic components can be viewed as autonomous agents, which leads to the notion of mechatronic systems as multi-agent systems. A suite of plug-ins that is currently under development aims to enable the prototyping and verification of such systems by extending existing modeling and code generation tools.

The first area of interest is the prototyping of simulated versions of the agents' physical environment. The environment model and the agents' capabilities for perceiving and manipulating it are specified and generated based on UML class diagrams and story patterns [29].

Building on this model of the observable environment, the social structure of the system is modeled. A decomposition into communities of agents provides a separation of concerns and enables an intuitive specification of the agents' interactions and social dependencies using story patterns [31].

Finally, critical aspects of the system model will be model checked, while non-critical aspects are optimized using empirical evaluation [30]. The revised behavioral specification will then be implemented by individual agents, which we ultimately plan to derive automatically. [fl]

[2]Formerly known as *hybrid statecharts*.

5.6 Scenario-Based Synthesis

The scenario-based synthesis approach for parameterized real-time behavior presented in [22, 20] extends our approach for the compositional formal verification of UML-RT models described by components and patterns [24]. The scenario-based synthesis techniques facilitates the design and verification steps by automatically deriving the required pattern behavior. Starting from a set of timed scenarios, the presented procedure generates a set of Statecharts with additional real-time annotations that realize these scenarios. As parameterized timed scenarios are supported, different system configurations can be specified as required by adjusting the behavior using the specific timing constraints.

In the future work we plan to improve the approach by supporting parameters in an even more general manner. In addition, we want to look into hybrid behavior and opportunities to also do synthesis for parameterized real-time behavior with restricted continuous behavior. [hg]

5.7 Dependable Systems

Based on the above presented support for real-time and hybrid systems, support for modeling of dependable systems is added to the Fujaba Real-Time Suite. *Fault Tolerance Patterns* [50] capture the structure of standard fault tolerance techniques like distributed recovery blocks or triple modular redundancy setups. In addition, deployment restrictions and degradation rules are specified for these patterns. The patterns are easily applicable to component-based real-time systems and are used to satisfy required hazard likelihoods in combination with a compositional hazard analysis approach [25]. Based on the aforementioned deployment restrictions, viable deployments are automatically computed prior to the systems systems as well as online for repair. If a full repair of the system is not possible, degradation rules [49] are used to degrade the systems functional or non-functional properties in order to keep its operating.

Currently, we address the behavior synthesis of generic components in fault tolerance patterns like voter components. [mtt]

6. METHODOLOGY

6.1 The Fujaba Process

In order to facilitate the use and the teaching of Fujaba and story driven modeling, the Fujaba group at University of Kassel has developed tool support for the so-called *Fujaba Process*. The Fujaba Process is of course iterative, use-case driven and test-centered as a modern process should be. The XProM plug-in developed at University of Kassel provides dedicated tool support for this process. The process is organized in the following steps:

First, the developer organizes his functional requirements into use cases.

Second, each use-case is elaborated by a textual scenario descriptions using a predefined format.

Third, these textual use case scenarios are elaborated into so-called story boards. A story board is a simple activity diagram where the activities contain collaboration diagrams modeling the corresponding step in detail.

During editing the story boards, Fujaba collects declarations of new classes, associations, and attributes. Thus, at the end of the story boarding, a first conceptual class diagram has already been created.

In addition, the XProM plug-in turns story boards automatically into JUnit tests. These JUnit tests create the start situation of the story board, invoke the method that realizes the use case and compare the resulting object structure with the result situation of the story board.

In case the generated JUnit test fails, XProM starts the Dobs debugger [19], automatically. This allows to compare the actual object structure resulting from the test run with the desired object structure depicted in the story board. In current work, we extend this mechanism to protocol automatically, which intermediate story board steps have been accomplished and which are still open.

A code coverage tool will allow to identify code that is not employed in the JUnit test generated from the story boards resulting from the use case analysis. Such code is either dead or it covers special cases that are not yet contained in the use case scenarios.

The Fujaba process has been applied with great success in the programming methodologies courses at University of Kassel for three years now. It helps our students a lot to organize their work. The resulting project handbooks are so far of much higher quality than project documents created without this help in earlier courses. [az,cs]

6.2 Teaching Fujaba

Teaching informatics usually starts with an introduction into a programming language. Frequently, it is even restricted to programming language issues. Object orientation and OO modelling are considered to be too complicated for introductory courses. Thus, the tackling of OO concepts is postponed to university, a fact which we feel is deplorable and at the same time unnecessary. To improve this situation, we developed a new approach to teaching OO modelling in introductory informatics at secondary school level from the very first lesson. We first introduce objects and how to model simple situations with object diagrams. Then we show how to operate on and with object models in simple steps. Once these ideas are understood, the derivation of class diagrams is almost a matter of course. This is then systematically turned into OO programs always examining certain scenarios and certain situations and the behavior and interaction of certain objects. By using Fujaba we are able to teach OO and to develop executable models at the same time (see [16]). [id]

7. FUTURE WORK

7.1 OCL Integration

In cooperation with the TU Dresden, we aim to integrate the Dresden OCL Toolkit [18] into Fujaba. Currently, OCL support for Fujaba's class diagrams is developed. In Fujaba's story patterns, constraints are used frequently for attribute assertions and for general constraints. Such constraints are written down using plain Java and are then copied into the Java code. It would be more reasonable to also use OCL here. From such OCL constraints, Java code has to be generated using the Dresden OCL Toolkit again. This integration is planed as second step of the OCL integration. [lg]

7.2 Usecase Driven Component Segmentation

A project in cooperation with an industrial partner deals with the problem to answer the question: Is it more costly to re-engineer an existing system or to reimplement it. A constraint from the industrial partner is that the new software system should be constructed out of components and the components should be reused in other products also. A problem is the granularity of the component, because the reuse is based on use-cases and not on technical issues. Therefore in the project, we use the use-cases as starting point and try to assign parts of the existing software to the use-cases. Based on these assignments an algorithm will identify components by rearranging parts of the software. We will use cohesion and correlation metrics to value the computed component structure on the one hand. On the other hand, we use a cost-model for the rearrangement operations that will produce an overall 'price' of the restructuring. The decision of re-engineering or reimplementing the system should then be answer easier. [jn]

7.3 Adaptable and Component-Based Process Management System

We are currently investigating the application of Fujaba to process modeling and process management. Here, the term "process" refers to development processes in engineering disciplines - including, but not limited to software engineering. Software processes - which we will assume for the rest of this section - span a wide spectrum and vary considerably depending on factors such as the size of the organization, the domain to be covered (e.g., embedded systems vs. business applications), the maturity of process capabilities, etc. Furthermore, they are usually dynamic and may be planned in advance only to a limited extent (in contrast to routine business processes). Therefore, developing appropriate tools for managing software processes has proved to be a challenging task. In past research, we have designed and implemented AHEAD [52], an adaptable and human-centered environment for the management of development processes. Unfortunately, AHEAD is adaptable only to a limited extent: AHEAD can be adapted by (evolvable) process model definitions, but the underlying process meta model is fixed. Our current research is target towards a system which supports definition and customization of the process meta model. Furthermore, the scope of support should be customizable, as well (e.g., by interfacing with an existing software configuration management system rather than providing an own supporting component). The system will be designed and implemented with the help of Fujaba. Our intent is to structure it into a set of small and reusable components from which the desired process support may be configured. [bw]

REFERENCES

[1] Altheide et al. An Architecture for a Sustainable Tool Integration. In Dörr and Schürr, editors, *TIS 2003 Workshop on Tool Integration in System Development*, pages 29–32, 2003.

[2] C. Amelunxen. Building a MOF 2.0 Editor as Plugin for FUJABA. In A. Schürr and A. Zündorf, editors, *FUJABA Days 2004*, volume Technical Report tr-ri-04-253, pages 43–47. Universität Paderborn, 2004.

[3] C. Amelunxen, T. Rötschke, and A. Schürr. Graph

Transformations with MOF 2.0. In *FUJABA Days 2005*. Universität Paderborn, 2005. Accepted for publication.

[4] C. Atkinson and T. Kühne. Aspect-oriented development with stratified frameworks. *IEEE Software*, 20(1):81–89, January/February 2003.

[5] B. Becker. Automatischer nachweis von induktiven invarianten in unendlichen systemen. Bachelor thesis, University of Paderborn, Paderborn, Germany, 2005.

[6] 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.

[7] G. Bollella, B. Brosgol, S. Furr, S. Hardin, P. Dibble, J. Gosling, and M. Turnbull. *The Real-Time Specification for Java^{TM}*. Addison-Wesley, 2000.

[8] S. Burmester, M. Gehrke, H. Giese, and S. Oberthür. Making Mechatronic Agents Resource-aware in order to Enable Safe Dynamic Resource Allocation. In B. Georgio, editor, *Proc. of Fourth ACM International Conference on Embedded Software 2004 (EMSOFT 2004), Pisa, Italy*, pages 175–183. ACM Press, September 2004.

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

[10] S. Burmester and H. Giese. Visual Integration of UML 2.0 and Block Diagrams for Flexible Reconfiguration in Mechatronic UML. In *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, Texas, USA*, pages 1–8. IEEE Computer Society Press, September 2005. (accepted).

[11] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, May 2005.

[12] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. 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)*, 6(3):203–218, August 2004.

[13] S. Burmester, H. Giese, A. Seibel, and M. Tichy. Story Patterns for Hard Real-Time Systems. In *Proc. of the Fujaba Days 2005, Paderborn, Germany*, September 2005. submitted.

[14] S. Burmester, H. Giese, and M. Tichy. Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In U. Assmann, A. Rensink, and M. Aksit, editors, *Model Driven Architecture: Foundations and Applications*, volume 3599 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, 2005. to appear.

[15] J. N. C. Schneider, A. Zündorf. Coobra - a small step for development tools to collaborative environments. In *Workshop on Directions in Software Engineering Environments; 26th international conference on software engineering, Scotland, UK*, 2004.

[16] I. Diethelm, L. Geiger, and A. Zündorf. Teaching modeling with objects first. In *Proc. of WCCE 2005, 8th World Conference on Computers in Education, Cape Town, South Africa*, July 2005.

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

[18] T. Dresden. Dresden ocl toolkit. http://dresden-ocl.sourceforge.net/, 2005.

[19] L. Geiger. Design level debugging mit fujaba. Bachelor thesis, Technical University of Braunschweig, Germany, 2002.

[20] H. Giese and S. Burmester. Analysis and Synthesis for Parameterized Timed Sequence Diagrams. In H. Giese and I. Krüger, editors, *Proc. of the 3rd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (ICSE 2003 Workshop W5S), Edinburgh, Scotland*, pages 43–50. IEE, May 2004.

[21] 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*, pages 179–188. ACM Press, November 2004.

[22] H. Giese, F. Klein, and S. Burmester. Pattern Synthesis from Multiple Scenarios for Parameterized Real-Timed UML Models. In S. Leue and T. Systä, editors, *Scenarios: Models, Algorithms and Tools*, volume 3466 of *Lecture Notes in Computer Science*, pages 193–211. Springer Verlag, April 2005.

[23] H. Giese and D. Schilling. Towards the Automatic Verification of Inductive Invariants for Invinite State UML Models. Technical Report tr-ri-04-252, University of Paderborn, Paderborn, Germany, December 2004.

[24] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11)*, pages 38–47. ACM Press, September 2003.

[25] 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*, volume 3219 of *Lecture Notes in Computer Science*. Springer Verlag, September 2004.

[26] J. Jahnke and A. Zündorf. Rewriting poor design patterns by good design patterns. In S. Demeyer and H. Gall, editors, *Proc. of the ESEC/FSE Workshop on Object-Oriented Re-engineering, Paderborn, Germany*. Technischer Bericht TUV-1841-97-10, Technical University of Vienna, Information Systems Institute, Distributed Systems Group, September 1997.

[27] U. Kelter, J. Wehren, and J. Niere. A generic difference algorithm for uml models. In *Proc. of the Software Engineering Conference (SE2005), Essen, Germany*, March 2005.

[28] F. Klar. SPin – Ein Werkzeug zur Realisierung von

Architektur-Stratifikation. Diplomarbeit, April 2005.

[29] F. Klein and H. Giese. Ontologiebasiertes Rapid Prototyping für kognitive Multiagentensysteme. In *Modellierung 2004 - Praktischer Einsatz von Modellen, Workshop W4: Ontologien in der und für die Softwaretechnik, Marburg, 2004*, pages 33–42. Conradin Verlag, Marburg, March 2004.

[30] F. Klein and H. Giese. Analysis and Design of Physical and Social Contexts in MultiAgent Systems using UML. In R. C. et al., editor, *Proc. of the 4th Workshop on Software Engineering for Large-Scale Multi-Agent Systems (in Conjunction with the International Conference on Software Engineering), St. Louis, MO, USA*, pages 1–7. IEEE, May 2005. accepted.

[31] F. Klein and H. Giese. Separation of concerns for mechatronic multi-agent systems through dynamic communities. In R. Choren, A. Garcia, C. Lucena, and A. Romanovsky, editors, *Software Engineering for Multi-Agent Systems III: Research Issues and Practical Applications*, volume 3390 of *Lecture Notes in Computer Science*, pages 272–289. Springer Verlag, February 2005.

[32] Königs and Schürr. MDI - a Rule-Based Multi-Document and Tool Integration Approach. *Special Section on Model-based Tool Integration in Journal of Software&System Modeling*, 2005. submitted for publication.

[33] M. Matula. *NetBeans Metadata Repository*. SUN Microsystems, März 2003.

[34] M. Meyer and J. Niere. Calculation and visualization of software product metrics. In *Proc. of the 3nd Fujaba Days, Paderborn, Germany*, September 2005.

[35] M. Meyer and L. Wendehals. Teaching object-oriented concepts with eclipse. In *Proc. of the Eclipse Technology eXchange Workshop (ETX), Satellite Event of the* 19^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Vancouver, Canada*, pages 1–5. ACM Press, October 2004.

[36] Mosher. *A New Specification for Managing Metadata*. Sun Microsystems, 2002.

[37] J. Niere. Visualizing differences of uml diagrams with fujaba. In *Proc. of the 2nd Fujaba Days, Darmstadt, Germany*, October 2004.

[38] J. Niere, W. Schäfer, J. P. 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.

[39] J. Niere, J. P. Wadsack, and L. Wendehals. Handling Large Search Space in Pattern-Based Reverse Engineering. In *Proc. of the* 11^{th} *International Workshop on Program Comprehension (IWPC), Portland, USA*, pages 274–279. IEEE Computer Society Press, May 2003.

[40] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification*, March 2003. ptc/03-10-04.

[41] D. Ohst, M. Welle, and U. Kelter. Difference Tools for Analysis and Design Documents. In *Proceedings of the IEEE International Conference on Software Maintenance 2003 (ICSM2003), Amsterdam*, pages 13–22, September 2003.

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

[43] OMG. *Request for Proposal: MOF 2.0 Query/Views/Transformations RFP*, 2002. http://www.omg.org/cgi-bin/doc?ad/2002-04-10.

[44] QVT-partners. *QVT-partners revised submission to QVT*, 2003. http://qvtp.org/downloads/1.1/qvtpartners1.1.pdf.

[45] I. Rockel. Versionierungs- und mischkonzepte für uml diagramme. Master's thesis, University of Paderborn, Germany, 2000.

[46] T. Rötschke. Adding Pluggable Meta Models to FUJABA. In A. Schürr and A. Zündorf, editors, *FUJABA Days 2004*, volume Technical Report tr-ri-04-253, pages 57–62. Universität Paderborn, 2004.

[47] T. Rötschke. Re-engineering a Medical Imaging System Using Graph Transformations. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, volume 3062 of *Lecture Notes in Computer Science*, pages 185–201. Springer, 2004.

[48] A. Schürr. Specification of graph translators with triple graph grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science,* 20^{th} *International Workshop, WG '94*, volume 903 of *LNCS*, pages 151–163, Herrsching, Germany, June 1994.

[49] M. Tichy and H. Giese. Extending Fault Tolerance Patterns by Visual Degradation Rules. In *Proc. of the Workshop on Visual Modeling for Software Intensive Systems (VMSIS) at the the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, Texas, USA*, September 2005. (accepted).

[50] M. Tichy, H. Giese, D. Schilling, and W. Pauls. Computing Optimal Self-Repair Actions: Damage Minimization versus Repair Time. In R. de Lemos and A. Romanovsky, editors, *Proc. of the ICSE 2005 Workshop on Architecting Dependable Systems, St. Louis, Missouri, USA*. ACM Press, May 2005.

[51] 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.

[52] B. Westfechtel. Ein graphbasiertes Managementsystem für dynamische Entwicklungsprozesse. *Informatik Forschung und Entwicklung*, 16:125–144, 2001.

[53] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. Habilitation Thesis.

APPENDIX

A. AUTHORS

ak: Alexander Königs, Real-Time Systems Lab, Darmstadt University of Technology, Germany, `alexander.koenigs@es.tu-darmstadt.de`

as: Andy Schürr, Real-Time Systems Lab, Darmstadt University of Technology, Germany, `andy.schuerr@es.tu-darmstadt.de`

az: Albert Zündorf, Software Engineering Research Group, University of Kassel, Germany,
albert.zuendorf@uni-kassel.de

bw: Bernhard Westfechtel, Applied Computer Science 1, University of Bayreuth, Germany,
bernhard.westfechtel@uni-bayreuth.de

ca: Carsten Amelunxen, Real-Time Systems Lab, Darmstadt University of Technology, Germany,
carsten.amelunxen@es.tu-darmstadt.de

cr: Carsten Reckord, Software Engineering Research Group, University of Kassel, Germany,
carsten.reckord@uni-kassel.de

cs: Christian Schneider, Software Engineering Research Group, University of Kassel, Germany,
christian.schneider@uni-kassel.de

ds: Daniela Schilling, Software Engineering Group, University of Paderborn, Germany,
das@uni-paderborn.de

fk: Felix Klar, Darmstadt University of Technology, Germany,
felix@klarEntwickelt.de

fl: Florian Klein, Software Engineering Group, University of Paderborn, Germany,
fklein@uni-paderborn.de

hg: Holger Giese, Software Engineering Group, University of Paderborn, Germany,
hg@uni-paderborn.de

id: Ira Diethelm, Software Engineering Research Group, University of Kassel, Germany,
ira.diethelm@uni-kassel.de

jn: Jörg Niere, Software Engineering Group, University of Siegen, Germany,
joerg.niere@uni-siegen.de

lg: Leif Geiger, Software Engineering Research Group, University of Kassel, Germany,
leif.geiger@uni-kassel.de

lw: Lothar Wendehals, Software Engineering Group, University of Paderborn, Germany,
lowende@uni-paderborn.de

mg: Martin Girschick, Darmstadt University of Technology, Germany,
girschick@informatik.tu-darmstadt.de

mh: Martin Hirsch, Software Engineering Group, University of Paderborn, Germany,
mahirsch@uni-paderborn.de

mm: Matthias Meyer, Software Engineering Group, University of Paderborn, Germany,
mm@uni-paderborn.de

mtt: Matthias Tichy, Software Engineering Group, University of Paderborn, Germany,
mtt@uni-paderborn.de

rw: Robert Wagner, Software Engineering Group, University of Paderborn, Germany,
wagner@uni-paderborn.de

sb: Sven Burmster, Software Engineering Group, University of Paderborn, Germany,
burmi@uni-paderborn.de

tk: Thomas Kühne, FG Metamodeling, Darmstadt University of Technology, Germany,
kuehne@informatik.tu-darmstadt.de

tm: Thomas Maier, Software Engineering Research Group, University of Kassel, Germany,
thomas.maier@uni-kassel.de

tr: Tobias Rötschke, Real-Time Systems Lab, Darmstadt University of Technology, Germany,
Tobias.Roetschke@es.tu-darmstadt.de

B. PLUG-IN OVERVIEW

Name	Description	Contact	Status	Version	Domain	Fujaba	Dependencies
DifferenceAlgorithmPlugin	Difference algorithm configured for class diagrams	Jörg Niere <joerg.niere@uni-siegen.de>	released	1.1.0	Difference Support	4.x	
DifferenceViewerPlugin	Difference class diagram visualization	Jörg Niere <joerg.niere@uni-siegen.de>	released	1.1.0	Difference Support	4.3	DifferenceCalculatorPlugin
DifferenceIntegratorPlugin	Comfortable handling differences of class diagrams	Jörg Niere <joerg.niere@uni-siegen.de>	released	1.1.0	Difference Support	4.3	DifferenceCalculatorPlugin, DifferenceViewerPlugin

Figure 2: Diagram difference plug-ins

Name	Description	Contact	Status	Version	Domain	Fujaba	Dependencies
MOFLON/Editor	MOF 2.0 meta model editor	Tobias Rötschke <tobias.roetschke@es.tu-darmstadt.de>	alpha	0.9.1	Meta Modelling	5.0	
MOFLON/Compiler	JMI code generator for MOF 2.0 meta models	Carsten Amelunxen <carsten.amelunxen@es.tu-darmstadt.de>	alpha	0.9.1	Meta Modelling	5.0	MOFLON/Editor
MOFLON/RoseXMI	Rational Rose XMI import for MOF 2.0 meta models	Tobias Rötschke <tobias.roetschke@es.tu-darmstadt.de>	alpha	0.9.1	Meta Modelling	5.0	MOFLON/Editor
MOFLON/JConstraint	Java Constraints for MOF 2.0 meta models	Tobias Rötschke <tobias.roetschke@es.tu-darmstadt.de>	alpha	0.9.1	Meta Modelling	5.0	MOFLON/Editor
MOFLON/OCL	OCL Constraints for MOF 2.0 meta models	Carsten Amelunxen <carsten.amelunxen@es.tu-darmstadt.de>	planned	N/A	Meta Modelling	5.0	MOFLON/Editor, MOFLON/Compiler
MOFLON/TGG	Triple Graph Grammars for MOF 2.0 meta models	Alexander Königs <alexander.koenigs@es.tu-darmstadt.de>	planned	N/A	Meta Modelling	5.0	MOFLON/Editor, MOFLON/Compiler, MOFLON/OCL
SPin	Realization of Architecture Stratification - Navigate between abstraction levels of a system description	Felix Klar <felix@klarEntwickelt.de>	released	1.4	Model Transformation	4.3.2	
TGGEditor	Editor and code generator for triple graph grammar specifications	Robert Wagner <wagner@upb.de>	released	1.0.0	Model Transformation	4.3.2	
MoTE	Engine for model transformations using triple graph grammar specifications	Robert Wagner <wagner@upb.de>	alpha	0.1.0	Model Transformation	4.3.2	TGGEditor
CoMa	A consistency management plugin for the specification and execution of consistency rules	Robert Wagner <wagner@upb.de>	released	1.0.0	Model Transformation	4.3.2	

Figure 3: (Meta-)Model Driven Software Development plug-ins

Name	Description	Contact	Status	Version	Domain	Fujaba	Dependencies
Java AST	An abstract syntax tree (AST) for Java	Lothar Wendehals <lowende@upb.de>	released	1.1.1	Reverse Engineering	4.2	
Java Parser	A parser for Java source files	Lothar Wendehals <lowende@upb.de>	released	3.1.1	Reverse Engineering	4.3	Java AST 1.1
Pattern Specification	Pattern specification for static pattern recognition	Lothar Wendehals <lowende@upb.de>	released	2.1.1	Reverse Engineering	4.2	
Inference Engine	Inference engine for static pattern recognition	Lothar Wendehals <lowende@upb.de>	released	2.1.1	Reverse Engineering	4.2	Java AST 1.0
Pattern Recognition Engines Generator	Recognition engines generator for static pattern recognition	Lothar Wendehals <lowende@upb.de>	released	1.1.1	Reverse Engineering	4.2	Pattern Specification 2.0, Inference Engine 2.0
Inference Engine Statistics	Statistic evaluation of static pattern recognition	Lothar Wendehals <lowende@upb.de>	alpha	0.1.0	Reverse Engineering	4.3	Inference Engine 2.1
Association Detection	Association detection for UML class diagrams	Lothar Wendehals <lowende@upb.de>	released	1.0.0	Reverse Engineering	4.3	Java Parser 3.1, Inference Engine 2.1
Java Tracer	Tracing of Java programs	Lothar Wendehals <lowende@upb.de>	beta	1.0.1	Reverse Engineering	4.0	
Behavioral Pattern Specification	Behavioral pattern specification for dynamic pattern recognition	Lothar Wendehals <lowende@upb.de>	alpha	0.1.0	Reverse Engineering	4.3	Pattern Specification 2.1, UML Sequence Diagrams 0.2
Dynamic Design Pattern Recognition	Dynamic analysis for dynamic pattern recognition	Lothar Wendehals <lowende@upb.de>	alpha	0.1.0	Reverse Engineering	4.3	Inference Engine 2.1
Behavioral Patterns Engines Generator	Behavioral engines generator for dynamic pattern recognition	Lothar Wendehals <lowende@upb.de>	alpha	0.1.0	Reverse Engineering	4.3	Behavioral Pattern Specification 0.1, Dynamic Design Pattern Recognition 0.1
MetricCalculation	Calculation of software product metrics	Matthias Meyer <mm@upb.de>	alpha	0.1.0	Reverse Engineering	4.3	JavaAST 1.1, JavaParser 3.1
PolymetricViews	Polymetric views editor and generator	Jörg Niere <joerg.niere@uni-siegen.de>	alpha	0.0.1	Reverse Engineering	4.3	MetricCalculation
SPin / VMSynchronizer	constructs UML Classes out of Java Byte Code	Felix Klar <felix@klarEntwickelt.de>	released	1.4	Reverse Engineering	4.3.2	

Figure 4: Reverse Engineering plug-ins

Name	Description	Contact	Status	Version	Domain	Fujaba	Dependencies
RealtimeStatechart	A real-time extension for UML state machines	Sven Burmester <burmi@upb.de>	released	1.0.0	Real-Time	4.3.2	
RealtimeStatechartRealtimeJavaCodeGenerator	A code generator for Real-Time Statecharts generating Real-Time Java	Sven Burmester <burmi@upb.de>	released	1.0.0	Real-Time	4.3.2	RealtimeStatechart 1.0
UMLModelchecking	Main model checking plugin	Martin Hirsch <mahirsch@upb.de>	released	1.0.0	Real-Time	4.3.2	UMLRT 1.0
UMLModelchecking-UMLRT2	Main model checking plugin	Martin Hirsch <mahirsch@upb.de>	alpha	0.0.1	Real-Time	4.3.2	UMLRT2 2.0
UppaalPlugin	Backend for UMLModelchecking plugin	Martin Hirsch <mahirsch@upb.de>	released	1.0.0	Real-Time	4.3.2	UMLModelchecking
UppaalPlugin-UMLRT2	Backend for UMLModelchecking plugin	Martin Hirsch <mahirsch@upb.de>	alpha	0.0.1	Real-Time	4.3.2	UMLModelchecking-UMLRT2
RavenPlugin	Backend for UMLModelchecking plugin	Alexander Stecker <piwi@upb.de>	alpha	1.0.0	Real-Time	4.3.2	UMLModelchecking
InvariantChecking	Plugin for checking inductive invariants on story patterns	Basil Becker <basilb@upb.de>, Daniela Schilling <das@upb.de>	alpha	1.0.0	Real-Time	4.3.2	
UMLRT	Components and Pattern	Sven Burmester <burmi@upb.de>, Martin Hirsch <mahirsch@upb.de>	released	1.0.0	Real-Time	4.3.2	RealtimeStatechart 1.0
UMLRT2	Components and Pattern	Sven Burmester <burmi@upb.de>, Martin Hirsch <mahirsch@upb.de>	beta	2.0.0	Real-Time	4.3.2	RealtimeStatechart 1.0
EmbeddedStoryDiagram	Extensions to derive WCETs from story diagram specifications	Andreas Seibel <aseibel@upb.de>	alpha	1.0.0	Real-Time	4.3.2	
UMLRT2CppCodeGeneration	C++ code generation for class diagrams and story diagrams	Andreas Seibel <aseibel@upb.de>	alpha	1.0.0	Real-Time	4.3.2	EmbeddedStoryDiagram 1.0
HybridComponent	Hybrid components, hybrid reconfiguration charts, and C++ code generation	Sven Burmester <burmi@upb.de>	alpha	0.1.0	Real-Time	4.3.2	UMLRT2 2.0, UMLRT2CppCodeGeneration 1.0
DeploymentBase	Base plugin for all deployment related plugin.	Matthias Tichy <mtt@upb.de>	alpha	1.0.0	Real-Time	4.3.2	UMLRT2 2.0
DeploymentTemplates	Support for fault tolerance patterns and their deployment restrictions.	Matthias Tichy <mtt@upb.de>	alpha	1.0.0	Real-Time	4.3.2	DeploymentBase 1.0
DeploymentConstraints	Specification of deployment constraints for individual components.	Matthias Tichy <mtt@upb.de>	alpha	1.0.0	Real-Time	4.3.2	DeploymentBase 1.0
DeploymentEnvironment	Specification for hardware ressources for deployment.	Matthias Tichy <mtt@upb.de>	alpha	1.0.0	Real-Time	4.3.2	DeploymentBase 1.0
ConcreteDeployment	Plugin for concrete deployment of software components to hardware ressources.	Matthias Tichy <mtt@upb.de>	alpha	1.0.0	Real-Time	4.3.2	DeploymentEnvironment 1.0
SceBaSy	Synthesizes real-time patterns from sequence diagrams with real-time annotations.	Sergej Tissen <serti@upb.de>	alpha	1.0.0	Real-Time	4.3.2	UMLSequenceDiagrams, UMLRT2

Figure 5: Real-time and mechatronic plug-ins

Name	Description	Contact	Status	Version	Domain	Fujaba	Dependencies
AspectF	AspectJ for Fujaba	Christian Schneider <christian.schneider@uni-kassel.de>	released	0.1	Aspect Oriented Programming	4.3	
CoObRA Plugin	Generate CoObRA support for the modeled application	Christian Schneider <christian.schneider@uni-kassel.de>	released	1.1	Persistency	4.3	Dobs
Dobs	Dynamic Object Browsing System	Leif Geiger <leif.geiger@uni-kassel.de>	released	3.3	Visualization / Debugging	4.3	
EditModes	Alternate editing mode for Fujaba Diagrams (dialogless, Classdiagrams only atm)	Christian Schneider <christian.schneider@uni-kassel.de>	alpha	0.1	General	4.3	
jEdit	Replace Fujaba's text editor (MPEdit) by jEdit	Christian Schneider <christian.schneider@uni-kassel.de>	released	0.2	General	4.3	
Refactorings	Provide refactorings such as extract method, change signature, implement/override method, ...	Christian Schneider <christian.schneider@uni-kassel.de>	released	0.1	General	4.3	
SVGGen	Print Fujaba diagrams to svg, pdf, png, ...	Matthias Tichy <mtt@upb.de>	released	1.1	General	4.3	
XProM	eXtreme Project Management	Leif Geiger <leif.geiger@uni-kassel.de>	released	2.1	Process Management	4.3	XMLReflect, Dobs, UseCasePlugin
UML Sequence Diagrams	UML sequence diagrams	Lothar Wendehals <lowende@upb.de>	alpha	0.2.0	General	4.2	
UseCasePlugin	UML use case diagrams	Jörg Niere <joerg.niere@uni-siegen.de>	beta	0.9.9	General	4.3	
XMISupportPlugin	Import and export of class diagrams	Jörg Niere <joerg.niere@uni-siegen.de>	released	1.0.0	General	4.3	
ConsolePlugin	Simple console window	Jörg Niere <joerg.niere@uni-siegen.de>	released	1.0.3	General	4.0	
BlockDiagramEditor	Editor for SDL block diagram specifications	Robert Wagner <wagner@upb.de>	released	1.0.0	Flexible Production Systems	4.3.2	
PLCCodeGenerator	Generates code from statecharts for programmable logic controllers	Robert Wagner <wagner@upb.de>	alpha	0.0.1	Flexible Production Systems	4.3.2	CoMa, MoTE

Figure 6: Other plug-ins

Keynote Talk:
Recent Trends in MDE: Principles, Standards, Platforms and Applications

Jean Bézivin
ATLAS Group (INRIA & LINA)
University of Nantes
2, rue de la Houssinière
44322 Nantes cedex 3
France
Jean.Bezivin@univ-nantes.fr

ABSTRACT
Several initiatives like the European ModelWare Integrated Project are currently investigating the applicability of Model Driven Engineering (MDE) to change the current practices of software development and maintenance. As an emerging area, MDE is still very much in evolution. The industrial demand is quite high while the research answer for a sound set of foundation principles is still far from being stabilized. Since the MDA™ initial proposal by OMG five years ago, MDE has much changed and gained in consideration but still has to prove that it is really leading to a major long-term technological evolution. To this end, what is much needed is a precise definition of the MDE goals and non-goals and a stable statement of the set of principles from which MDE foundations could be assessed. One important question is how MDE relates to other contemporary technologies (Object Technology, XML, etc.). Another way to provide a characterization of MDE is by listing the various operations like transformations that could be performed on basic models. The talk will discuss the historical contributions of MDE, its present state and potential future evolutions as well as related open research issues.

SPin – A Fujaba Plugin for Architecture Stratification

Felix Klar, Thomas Kühne, Martin Girschick
Fachgebiet Metamodellierung
Fachbereich Informatik
Technische Universität Darmstadt, Germany
felix@klarentwickelt.de, {kuehne, girschick}@informatik.tu-darmstadt.de

ABSTRACT

SPin is a plugin for Fujaba that provides basic support for architecture stratification. It enables Fujaba models to be annotated with refinement directives which may then automatically be executed by the plugin. The corresponding refinement transformations may be defined with a combination of *story driven modeling* and Java coding. These transformations affect both model and associated code, and may be defined interactively, i.e., do not require Fujaba to be shutdown and started up again. In this paper we describe the purpose of the plugin, how to use it, its realization, and some supporting functionality.

Keywords

stratification, model driven development, model transformation, Fujaba plugin

1. INTRODUCTION

Today's software systems have reached such a level of complexity that a single view, e.g., architectural description, is not sufficient anymore. If the system is described from a bird's eye view, using a very high level architecture description, many important details regarding performance, extensibility, etc. remain hidden. If, however, one chooses a view with a much lower level of abstraction, allowing the above properties to be evaluated, the complexity will become unwieldy; it becomes difficult to see the forest for the trees.

Architecture stratification is an approach that connects multiple views on a single system with refinement translations, so that each view describes the whole system on a particular level of abstraction. This way single levels do not only present an optimal mix of overview and detail for various stakeholders, but they also separate and organize a system's extension points, patterns, and concerns [1].

The Fujaba plugin SPin[1] [4] supports the automatic transformation of models into more detailed versions and thus represents basic support for architecture stratification. However, it is not restricted to this particular flavor of model driven development, but supports any development approach that requires annotation-guided model transformations.

In the following, we first describe SPin from a user's perspective (section 2) and then present an example demonstrating the utility of SPin (section 3). Subsequently we take a closer look at the inside of SPin (section 4), before we outline future work (section 6) and finally conclude (section 7).

[1]An acronym for "**S**tratification **P**lug**in**".

2. USING SPin

Figure 1 shows how SPin may be used in the context of Fujaba. SPin supports two kinds of transformations, refinement rules, yielding more concrete models and abstraction rules, yielding more abstract models both of which can be regarded as endogenous transformations [5]. In the following we will concentrate on rules defining refinement transformations only, though.

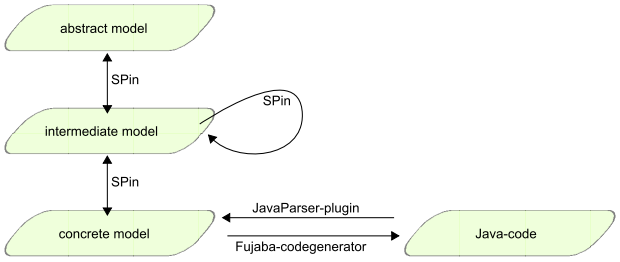

Figure 1: *SPin's role within Fujaba*

Since SPin's transformation rules may not only transform models (e.g., class diagrams) but also any associated code (e.g., method implementations) it can be used to transform a simple system description into a complex one, using multiple steps. The most complex system description can then be used to create an executable system by virtue of the Fujaba code generation engine.

The prerequisite for automatically transforming models in this top-down fashion, however, are directives, i.e., annotations in a model.

2.1 Annotating a model

Annotations specify in which way a model element should be refined in order to obtain a finer grained realization. To provide additional information for the transformation, annotations can be parameterized using basic types (e.g., a string specifying the name of a class that should be generated) or links to other model elements (e.g., specifying one or more of the already existing elements to be used as observers for a subject). We therefore chose a notation similar to UML collaborations in UML class diagrams. Both notations share

the need to specify which elements form a structure—such as which other element(s) should be involved in the transformation process or what other element to use as a parameter to the transformation—and the need to describe the role of the referenced element.

SPin provides a dedicated *annotation editor* to support the introduction and parameterization of annotations. Figure 2 shows a screenshot of the annotation editor displaying the parameters of an "Observer" annotation (see the example in section 3). The annotation is parameterized with two links ("state" and "concreteObserver") and one basic type ("observerClassName").

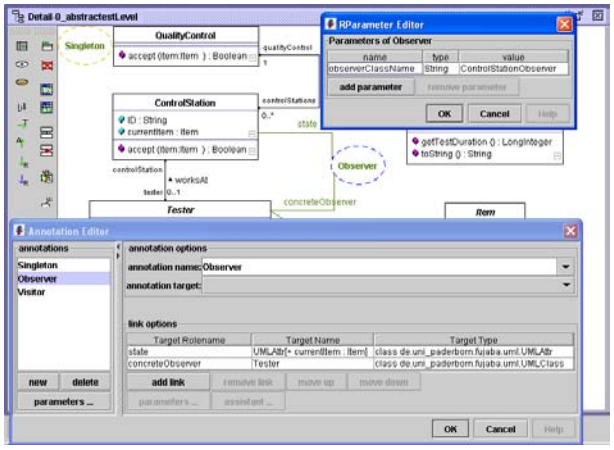

Figure 2: *SPin's annotation editor*

Once a model is completely annotated, the user may use the context menu of an annotation to initiate the corresponding transformation process. Currently, SPin supports manual transformation initiation only, i.e., it is not possible yet to start a recursive unfold process which stops when no more annotations exists.

2.2 Creating a rule

Unfolding an annotation triggers the corresponding refinement rule. Such rules are completely user defined. SPin only provides the machinery for creating, using, and executing rules. The rules themselves are part of a rule library, which can be extended dynamically.

Let us step through the creation of a rule implementing the GoF [3] pattern "singleton". First, we create a UML class diagram and then add a new rule class to it by invoking the "create rule..."-action from the class diagram's context-menu. This causes SPin's "new rule" dialog to open and we use it to specify the rule type and a rule name. In our example we choose a refinement rule with the name "Singleton". The rule's name describes its intent, but will also later be used to annotate a model.

Figures 3 and 4 show a part of what SPin automatically generates after the "new rule" dialog has been closed.

Figure 3 shows the addition of a new refinement class (*RRSingleton*). Among other features it defines an 'apply' method

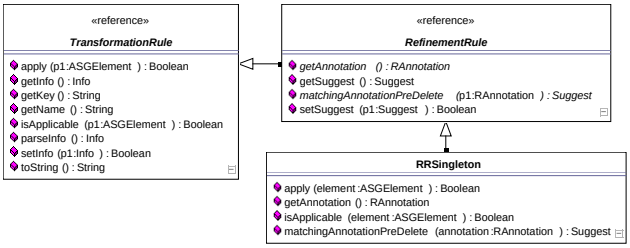

Figure 3: *A new refinement rule*

that contains the actions to be performed when the rule is triggered.

Figure 4 shows the automatically generated body for 'apply'. As one can see Fujaba's *Story Driven Modeling* (SDM) [6] is used to implement the 'apply'-method. This results in a semi-graphical implementation which is more self-explanatory and easier to create and to maintain than handwritten Java-Code. The first check makes sure that the model element to be transformed indeed has the correct annotation ("Singleton" in this case). If yes, a reference to a *UMLFactory* is created so that new UML elements may be created in the core transformation part. Finally the annotation is destroyed, i.e., removed from the diagram, since at this point in time the annotation has served its purpose to create a more concrete realization of its source structure. The rule designer may still change any part of this, but this is how most refinement rules look like, if one leaves out the core transformation part and any further checks as to whether the rule is really applicable.

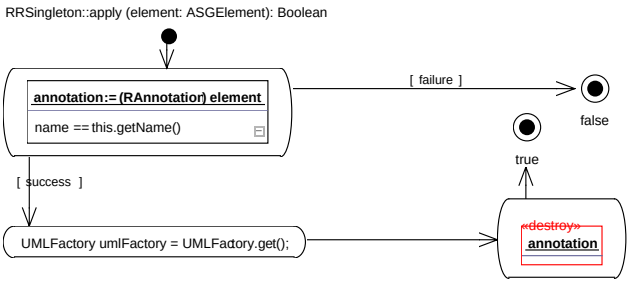

Figure 4: *Generated 'apply'-method*

In our example the rule's precondition has to be enhanced to check whether the annotation is bound to a UML class (see figure 5). If this is the case, the class will be transformed into a singleton class. The transformation code adds an attribute holding the singleton-instance, a private constructor and a get-method that returns the singleton-instance. Once finished, the rule can be exported to the rule library, so that it may be used to transform a UML class into a "singleton".

3. CASE STUDY

We now demonstrate the utility of SPin by considering an example system that simulates a quality control assembly line. In this example we use the three design-patterns "Sin-

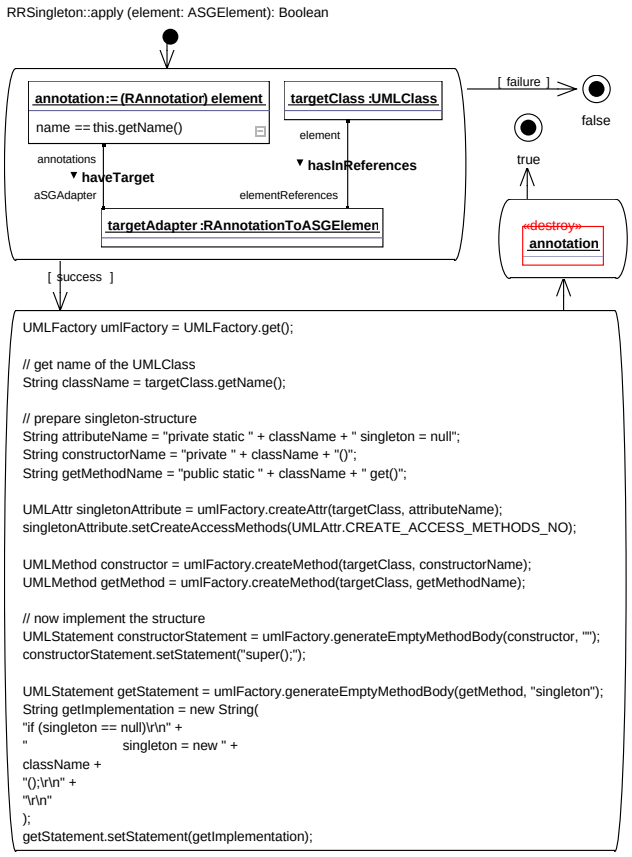

Figure 5: *Fully implemented 'apply'-method*

gleton", "Observer", and "Visitor" [3] in order to obtain a high-level view on the system's structure (see Figure 6).

3.1 System description

The system has a main quality control unit (*QualityControl*) that should be realized as a singleton instance (hence the corresponding annotation). Quality control is realized as an assembly line that consists of a variable number of control stations (class *ControlStation*). These stations check items (abstract class *Item*) passed to them by the assembly line. Our example features only one concrete item type (class *Screw*).

Control stations feature a tester which checks the current item. For each observed item a test report (class *ItemTest*) is created. Testers come in two kinds: manual testers, like humans, that are able to perform very complex tests and automatic testers, like industry-robots that are specialized for testing a single property of an item. Here, a robot (class *Scale*) is used, that checks an item's weight.

Let's have a closer look at the annotations "Observer" and "Visitor". Annotation "Observer" is parameterized with two links. Link "state" binds attribute *currentItem:Item* of class *ControlStation* to the annotation to define which state should be observed. Link "concreteObserver" binds class *Tester* which should observe the specified state. An additional parameter "observerClassName" of base-type *String* with value "ControlStationObserver" has been added to the annotation.[2] This parameter specifies the name of the generated observer interface. Annotation "Visitor" has two links as well: "element" specifies which class should be the element of the visitor pattern and "concreteVisitor" specifies which class should visit the element. For a more detailed description of the rules corresponding to "Observer" and "Visitor" please see [4].

3.2 Refining the system

We now refine this system, by unfolding annotations step by step, until we reach the most detailed system description.

After unfolding the "Singleton" annotation we may then unfold "Observer". Note that we have to attach/detach concrete observers to/from their subjects (in this case: class *ControlStation*), so that observers will be notified of state changes. As we want these code-fragments to be placed in method 'setTester(Tester)' in class *ControlStation*, we need to implement this method manually, so it will attach/detach tester instances accordingly. We also want to specify what should be done when an observer is updated by its subject. This is accomplished by adding a call to method 'createItemTest(Item)' within the method 'update(ControlStation)' in class *Tester*.

Finally, we resolve "Visitor". Of course we need to provide the code for the visit-methods in each concrete visitor. This is currently done manually after the transformation step, but alternatively one may also provide the method bodies as parameter values to a correspondingly defined "Visitor" rule.

All other pattern-related method bodies will be automatically generated of by the respective rules. The resulting system structure is visualized in figure 7.

3.3 Completing the system

After the system has been refined to its most detailed version, we now have to complete the implementation by filling in the missing method bodies.

We only need to deal with two methods in our example: (a) 'process(Item)' in class *ControlStation*, which has to notify its observers, if an item receives the focus of a control station and (b) 'createItemTest(Item)' in class *Tester*, which has to initialize the visit-process by invoking method 'accept(ItemVisitor)' on the passed *Item*-instance.

Now that the most detailed model has been completed, Fujaba's codegenerator can be used to generate executable code from it.

4. INSIDE SPin

The following section describes some of the internal aspects of SPin, in particular how SPin provides support for the creation of new transformation rules.

[2]However, this parameter is not visible in figure 6. It may only be seen or changed through the annotation editor (see figure 2).

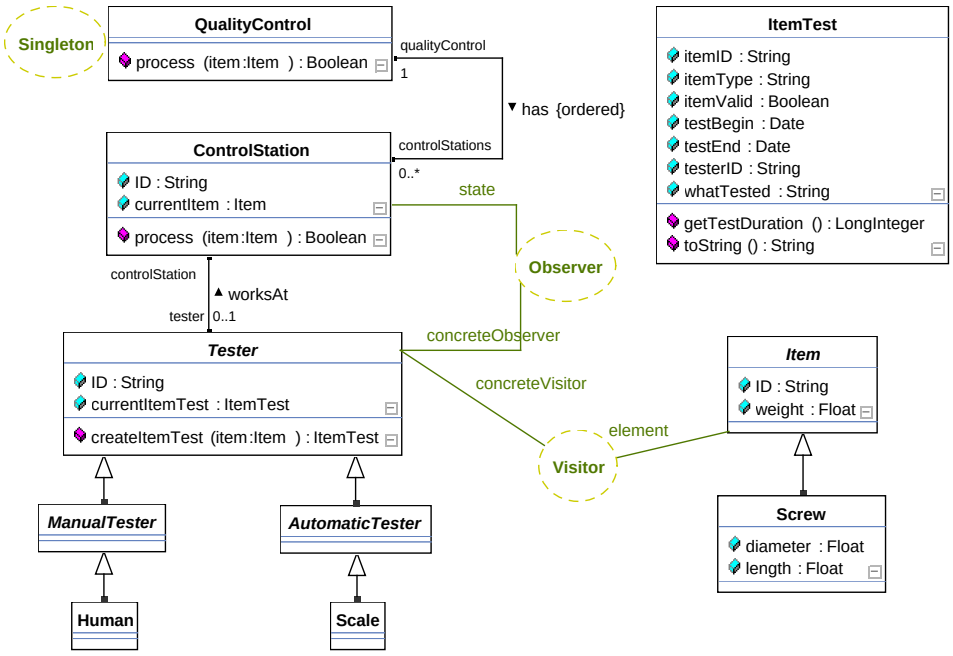

Figure 6: *Most abstract system version of our example*

4.1 Dynamic Rule Additions

Once a rule is created and added to the library, it can immediately be used in models without requiring manual shutdown, recompilation, etc. This is achieved in the following way: After creating or modifying a rule in the form of a UML class, the rule developer asks SPin to export the rule. SPin thereupon first employs Fujaba's code generator to generate the corresponding Java-code and then the Java compiler to create the respective Java bytecode. Subsequently, all new generated files are moved into the rule library directory and finally the library is reloaded.

4.2 Modifying Activity Diagrams

Often rules need to transform behavior in the form of method bodies. In Fujaba, method bodies are specified with storydiagrams, a combination of activity and collaboration diagrams [6].

In order to support the analysis and modification of such storydiagrams SPin introduces a standard mechanism (available through class *UMLActivityDiagramModifier*) that enables rule developers to visit each element of a storydiagram. It simply iterates over each element and invokes element-specific code on a visitor (class *UMLActivityDiagramVisitor*) that performs operations on those elements. This mechanism can easily be adapted by rule developers through implementing their own visitor classes and passing them to the diagram modifier. For instance, rule developers may use it to modify text within method bodies.

4.3 Element Creation and Initialization

A common activity of rules is the creation of new model elements. SPin provides factories (see *UMLFactory* in Figure 4), whose task is to reduce the effort for creating model elements to a minimum. In the ideal case just one function call is required to create and initialize an element and then add it to a diagram. SPin provides two factories: One supporting the creation of transformation rules and the other to create elements of the annotation metamodel.

One aspect of these factories—creation of elements—will be obsolete with Fujaba 5, when the creation of UML elements through factories will be natively supported. Yet, the other aspect—initialization of elements—will still be useful, since Fujaba 5 will still require manual initialization, requiring potentially many function calls. Hence, SPin's UML factory may even proof to be useful in Fujaba 5.

4.4 Metamodel Synchronization

One way to specify transformation rules is to use the SDM approach available in Fujaba [2]. Figure 4 shows how a story diagram is used to check the applicability of a rule. The transformation process itself may also be specified using SDM (instead of using Java).

However, in order to be able to employ SDM in this manner, Fujaba requires a UML classdiagram containing the appropriate metamodel to be present in the current Fujaba-project. The challenge is hence how to get this metamodel into a Fujaba-project, since one clearly does not want to do it manually. This would not only be a very time-comsuming but also a very error-prone task.

One solution to automate this process is the *JavaParser*-plugin (see also Figure 1). It can be used to generate a UML classdiagram from the Java sourcecode which represents the metamodel. If the source code and the actual metamodel built into Fujaba differ, though, e.g., because of a version mismatch, incompatibilities will arise.

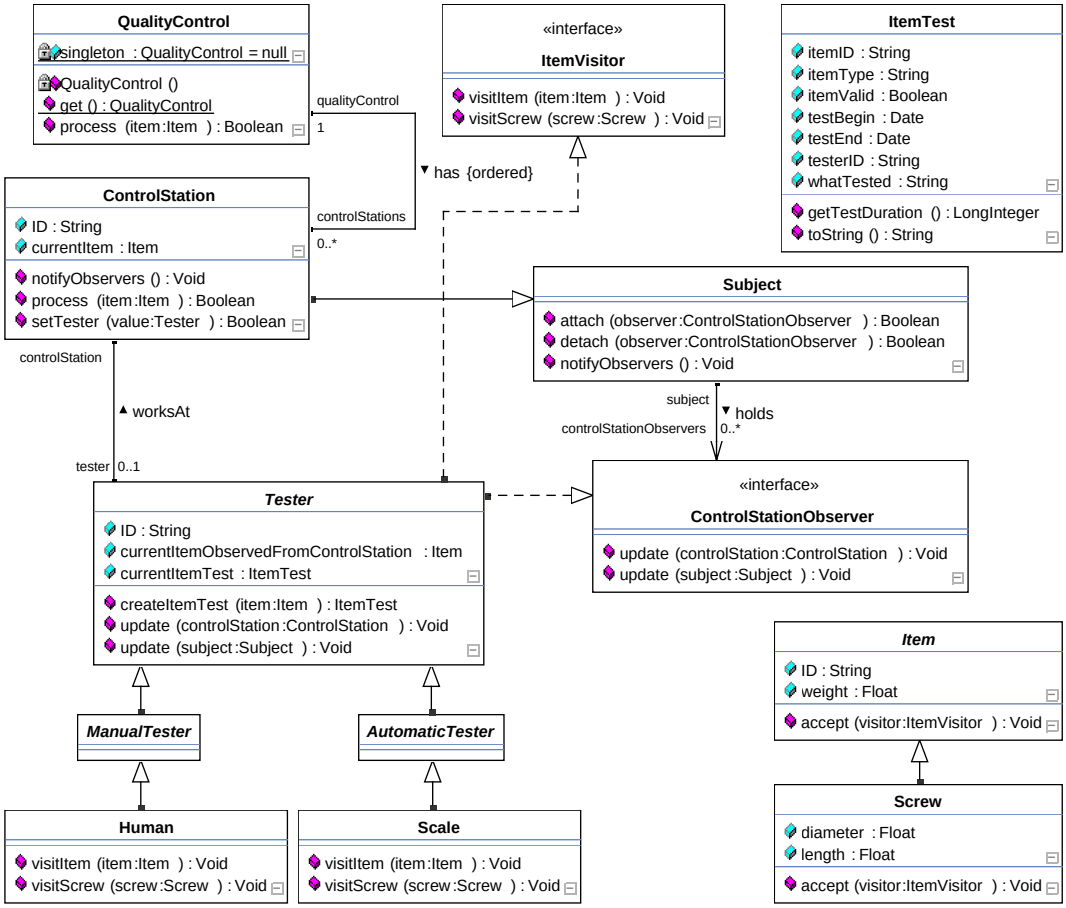

Figure 7: *Most detailed system version of our example*

SPin therefore uses a *Virtual Machine synchronizer* (VM-synchronizer) [4]. It takes the actual—and thus guaranteed to be of the correct version—Java bytecode of the respective part of Fujaba as input, extracts structural information using a Java classloader in conjunction with the Java reflection API and generates a UML model from the extracted information using a UML factory. Finally a post processing step detects associations between UML classes and adds them to the metamodel which then can subsequently be used by SDM.

5. RELATED WORK

Most MDA tools specialize on generating code from a model or migrating models in one modeling language to another, i.e., exogenous transformations [5]. Tools that support model refactorings can—according to [5]—be classified as supporting horizontal endogenous transformations. In contrast, architecture stratification realizes vertical endogenous transformations. In other words, refactorings maintain the same level of abstraction whereas architecture stratification creates different levels of abstraction expressed in the same modeling language.

Only few commerical tools, such as Together Architect[3] and ArcStyler[4] provide basic support for defining vertical endogenous transformations as well.

Together Architect provides an extendable template-based mechanism for creating patterns. This mechanism can be used to create rules that perform vertical endogenous transformations. A pattern manager is used to apply patterns to class diagram elements. In contrast to SPin, however, this way transformations are executed in a step by step fashion, whereas SPin automates the transformation of all annotations of one kind and will eventually support a fully automated application of all applicable transformations from top to bottom.

The MDA tool ArcStyler supports both model-to-model and model-to-code transformations, which are defined in so called *cartridges*[5]. A cartridge defines a source and target metamodel (or inherits the UML metamodel) and the transformations from the source to the target. UML stereotypes may be used to guide the transformation process. In addition so called *marks* are used to allow further parameterization of the model, e.g. for different target platforms. Transforma-

[3] http://www.borland.com/us/products/together/

[4] http://www.interactive-objects.com/

[5] http://www.interactive-objects.com/support/doc/doc/Carat_Guide.pdf

tions are described using the script language JPython. This is supplemented by the concept of blueprints which are similar to *model templates*. ArcStyler follows the MDA approach where a platform independent model (PIM) is completely parameterized and then transformed to a new platform specific model (PSM). If this approach is used in a staged, incremental manner, it very much resembles the abstraction level stratification approach of SPin.

Neither Together Architect nor ArcStyler support Fujaba's *Story Driven Modeling*, which is very useful for the semi-graphical specification of transformation rules as used in SPin.

6. FUTURE WORK

The current version of SPin offers a limited set of transformation rules. Although these are user extensible, the utility of SPin would be increased if it came with a rich set of ready-to-use rules. By applying the stratification process to big and complex software systems it will be possible to extract useful rules which can then be added to SPin.

Employing stratification in its intended form with SPin is currently hindered by the fact that only manual, stepwise initiations of transformations are supported. In order to fully automate the generation of a complex system from a given simple and abstract view, it is necessary to automate the process of unfolding annotations. This includes the specification of the order in which annotations are to be unfolded. However, this ordering is neither difficult to work out, nor should it be part of an automated process. Annotations exhibit natural dependencies and lend themselves to generate levels of system concerns [1]. It is therefore the task of the system architect to select which of the annotations are addressed at each specific abstraction level. As a result, future versions of SPin should provide a configuration system that allows users to specify and store their annotation processing orders.

SPin will significantly benefit from the new features of Fujaba 5. For instance, the then available support for multiple projects will enable developers to create rules in one project and immediately apply them in another. Moreover, users then might be able to easily navigate back and forth between different levels of abstraction.

Moving up in the hierarchy of abstraction levels is already possible as long as the more detailed versions were generated by SPin. This is accomplished by using Fujaba's built in "undo" mechanism.

In the current release of SPin the transformation rules are expressed using arbitrary Java code. This implies that there is no way to execute them "backwards" or to automatically generate inverse rules. The "reverse application" of forward directed generation rules is, however, an attractive facility for reverse engineering systems. This way, one may start from a complex system and simplify the system by either creating and applying "abstraction rules" or by using refinement rules in the "reverse" direction. This, however, requires rules that can either be transformed into their inverse or rules that work bi-directionally. Hence, we are planning to investigate replacing Java for specifying transformation rules with an approach that supports bi-directional rule application from the start, e.g., Triple Graph Grammars [7].

7. CONCLUSION

SPin is the result of a diploma thesis conducted at the metamodeling department of the Darmstadt University of Technology [4]. Although it currently provides only basic support for architecture stratification, it represents a starting point that can be extended into full stratification support.

Even the current SPin version, however, already demonstrates the feasibility of transforming both model elements and associated code in sync with each other, in order to obtain a fully specified complex system from a simple system, after a number of transformation steps. Current rule definitions sometimes require manual edits to the generated models—for user code, such as implementations of "visit" methods for the visitor pattern—however, this is no principle limitation of the approach. More sophisticated rules and corresponding annotation dialogs will allow the continuous and fully automated generation of models from top to bottom.

Transformation rules are user-definable, typically using a convenient mix of SDM (for pattern matching) and Java (for an unconstrained definition of transformations). Since SPin is able to dynamically integrate new rules, the development of the main system model and corresponding rules, can proceed in an interleaved and very interactive manner.

The creation of new rules in SPin is heavily assisted by a number of convenient utilities, such as support for modifying method bodies, element creation, and the synchronization of the UML metamodel.

In addition, the work on SPin resulted in a pattern-like notation for refinement annotations that enable transformation parameters to be specified both graphically (through labeled links) and non-graphically (through primitive parameter types entered into a corresponding dialog). This way one achieves fine grained control of the transformation process, using a concise notation.

We believe that SPin already represents an interesting starting point for supporting architecture stratification, but as we have outlined in the previous section on future work, we are convinced that it has an even higher potential for further development.

8. REFERENCES

[1] C. Atkinson and T. Kühne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1):81–89, 2003.

[2] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language and Java. Technical report, AG-Softwaretechnik, Fachbereich 17, Universität Paderborn, 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Entwurfsmuster*. Addison-Wesley, 1996.

[4] F. Klar. SPin – Ein Werkzeug zur Realisierung von Architektur-Stratifikation. Diplomarbeit, Technische Universität Darmstadt, April 2005, http://www.klarentwickelt.de/doc/science/diplomarbeit_klar.pdf.

[5] T. Mens, K. Czarnecki, and P. V. Gorp. A taxonomy of model transformations. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

[6] U. Nickel, J. Niere, and A. Zündorf. The FUJABA Environment. Technical report, Computer Science Department, University of Paderborn, 2000.

[7] A. Schürr. Specification of graph translators with triple graph grammars. Germany, June 1994. Herrschin, Springer Verlag. Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science.

Graph Transformations with MOF 2.0

Carsten Amelunxen, Tobias Rötschke, Andy Schürr
Technische Universität Darmstadt
Institut für Datentechnik, FG Echtzeitsysteme
Merckstr. 25
Darmstadt, Germany
[carsten.amelunxen|tobias.roetschke|andy.schuerr]@es.tu-darmstadt.de

ABSTRACT

The MOFLON framework aims to contribute to various aspects of meta model-based software engineering on top of the Fujaba Tool Suite. Currently, Fujaba operates on an internal UML 1.4-like meta model to represent graph schemata. Graph transformations are formulated in terms of this internal model. With the upcoming Fujaba 5 release, plugins will be able to provide their own meta models which are mapped onto the internal one. MOF 2.0 as the standard meta modeling language provides improved new concepts compared to older versions of UML and MOF. By implementing a MOF 2.0 editor and a related JMI compiler as plugins for Fujaba, we are now able to define graph schemata with MOF 2.0. In this paper, we discuss the implications of our choice of MOF 2.0 as schema language on the existing graph transformations in Fujaba and we sketch a roadmap how to adapt the Fujaba SDM editor and code generator appropriately.

1. INTRODUCTION

Meta modeling languages are used to describe other modeling languages. The Object Management Group (OMG) adopted MOF 2.0 [9] as standard meta modeling language. To be able to work with meta-modelled languages, tool providers need to map meta models on source code. Sun defined the Java Metadata Interface (JMI) [3] as standard mapping of MOF-compliant meta models to Java code. As we see interesting opportunities in the combination of MOF 2.0 as meta modelling language and Fujaba for graph transformations, we decided to bring both together.

During the last months, we have been busy on the first release of the MOF 2.0 plugin for Fujaba which, in more general terms, is a part of the MOFLON meta modeling framework[1], that had been proposed in [2] for the first time. On the last FujabaDays, we discussed the new association concept [1] of MOF 2.0 and how we can improve Fujaba so that the internal meta model used for graph transformations is separated from the external meta model defining the editor features visible to the user [12].

Meanwhile we have made major progres with the MOF 2.0 editor and the accompanied JMI code generator. We are able to bootstrap the creation of the underlying MOF 2.0 meta model. A MOF instance can be either directly created using the MOFLON/Editor plugin for Fujaba or drawn in Rational Rose, exported as XMI and finally imported in the plugin. Using our MOFLON/Compiler plugin, we can not only provide JMI interfaces, but also an implementation that is able to store a consistent model with respect to features like subsetting, redefinition, union, composition, and multiplicities.

[1]http://www.moflon.org

2. MOFLON INSIDE FUJABA

Figure 1 provides a sketch of the relationships between Fujaba and MOFLON. The MOFLON/Editor plugin realizes a MOF 2.0 meta model editor based on JMI-compliant MOF meta-meta model, and its adaption to the internal Fujaba meta model. The MOFLON/Compiler plugin adds a JMI-compliant Java code generator for MOF 2.0 instances created using the editor component. The MOFLON/RoseXMI plugin allows to import UML class diagrams from Rational Rose using XMI. During the import, a corresponding MOF 2.0 instance is created by our plugin, which can be further processed using the editor and compiler component.

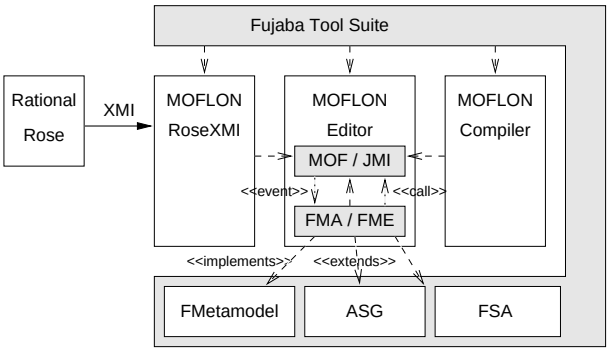

Figure 1: Sketch of MOFLON dependencies

The Fujaba framework provides the internal meta model (*FMetamodel*) together with a basic abstract implementation (*ASG*). The MOFLON/Editor plugin contains a generated JMI implementation of the MOF 2.0 meta model (*MOF*). This implemention fires MDR-like [7] events whenever a model instance is changed, and hence has no static dependencies on any Fujaba packages.

The internal Fujaba metamodel and the MOF metamodel are coupled via the Fujaba MOF Adapter (*FMA*) layer. Within the MOFLON/Editor plugin, every diagram item in terms of the internal Fujaba meta model is an adapter referring to a MOF element as adaptee (cf. Object Adapter

design pattern in [4], p. 141). For every adapter there is an adequate unparse module to visualize the item in the editor using the Fujaba Swing Adapter (*FSA*) layer. It must be mentioned that we proposed class adapters inheriting from *ASGElement* in [12] to implement the JMI interface, but we decided that our default implementation should be self-contained.

Method invocations to an adapter are directly delegated to the corresponding MOF element, which contains the actual state of the item. In the opposite direction, the Fujaba MOF EventListener (*FME*) package receives change events from the MOF implementation and fires the corresponding property change events to update the GUI.

In some cases (e. g. association ends, generalizations, mutual references), a number of adapters correspond to a single MOF element. In such a case, the adapters must remember their relationships with other adapters referring to the same MOF element. Besides, the adapters keep information related to those parts of the Fujaba meta model, that do not correspond to standard UML or MOF elements (e.g. projects, diagrams, files, and programming language-related data).

2.1 MOFLON vs. Fujaba

In the following the new features of MOFLON compared to Fujaba are listed, i.e. the delta from the Fujaba implementation of UML 1.4 class diagrams to MOF 2.0.

- **Packages** are namespaces in MOFLON and can be combined by two kinds of package merges[2]. Each package holds a diagram. The package concept replaces Fujaba's view diagrams.
- **Attributes** with collections as values can be subsetted, redefined, unique, ordered or an union of all their subsets. As a result of combining ordering and uniqueness, multi-valued association ends are realized by four different kinds of collections[3].
- **Associations** are distinguished into "real" associations and (mutual) references. References correspond to associations in Fujaba as they are mapped on attributes whereas associations are mapped on classes. Navigable association ends are treated like attributes of the appropriate classes. Thus the features concerning attributes are also relevant for navigable association ends. For the purpose of graph transformation both concepts of references and associations can be treated equally since there are no link objects for associations.
- **Navigability** specifies how association instances can be queried in MOFLON. Navigable association ends can be handled like attributes with the appropriate access methods. This easy access is only available for navigable association ends. Non-navigable association ends can be queried by using the methods of the class representing the association. By making use of the class for an association each link can be queried independently of the navigability. Associations are always navigable in terms of Fujaba.
- **Composition** is semantically relevant since all children of a composite are deleted together with their parent.
- **Multiplicities** are dynamically evaluated similar to composites. For a given object, multiplicity determines the required number of linked objects of the associated class. If the number of linked objects falls below the lower bound the given object is deleted, as in our opinion it cannot exist if the lower bound is violated. This evaluation is just processed with each deletion of an object since not every update can result in a consistent meta model instance. The demand of a consistent meta model instance after each deletion has been proven to be very useful. A more sophisticated approach is a matter of current work.

[2]There are two kinds of package merge (merge and combine). See [9] and [10] for details.
[3]Set, OrderedSet, Bag, Sequence

3. NEW FEATURES OF MOF 2.0

Figure 2 shows an example from the domain of reengineering in which we demonstrate some MOF 2.0 features that are new compared to UML 1.x and previous MOF versions. The *FileSystem* package defines the general concepts of UNIX-like file system with files and directories. The second package specializes the first one for the purpose of typical C programming.

Note, that we drew the diagrams for package contents inside the package to achieve a more compact visualization. In the MOFLON/Editor component, a dedicated diagram exists for every package. Package dependencies are drawn in the diagram of the parent package or in the root diagram for root packages.

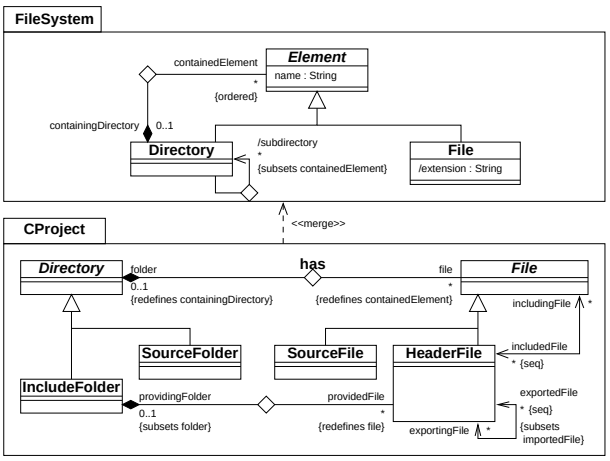

Figure 2: A sample MOF 2.0 diagram

To start with package relationships, consider the *merge* dependency between package *FileSystem* and *CProject*. This is one of the new package relationships, that are coming with MOF 2.0[4]. Every public Element[5] of package *FileSystem* is

[4]Other package relationships are *import* and *combine* which have different semantics
[5]*PackageableElement* to be exactly

automatically available in the namespace of package *CProject*. If a new element with an existing name is introduced, a generalization relationship to the existing element in the merged package is established by the semantics of the merge dependency. So *CProject::Directory* is a subclass of *FileSystem::Directory* and *CProject::File* is a subclass of *FileSystem::File*. As they are different elements, *CProject::File* may be abstract, while *FileSytem::File* is not. In case of a combine relationship the elements of *FileSystem* would have been (deep) copied into *CProject*. Equally named classes would have been merged instead of inherited. In case of an import relationship the elements of the imported package would have just been added to the namespace of the importing package.

Considering the revised definition of navigability in MOF 2.0 in combination with the relations between association ends, associations evolved to a more sophisticated concept. In MOFLON, associations are only allowed between classes and always drawn with a diamond to distinguish them from (mutual) references, like the relationship between *File* and *Headerfile*. References are a means to define "light-weight" unidirectional relationships between classes or data types and other classes without instantiating associations. Pairs of mutual references are treated as one "light-weight" bidirectional relationship. Taking into account that adornments like aggregation and composite have no semantics in Fujaba, these "light-weight" relationships correspond closely to Fujaba's association concept.

In UML 1.x, association ends are always unique, but in MOF 2.0, they may be non-unique as well. In combination with ordering, multi-valued association ends have to be realized by four different kinds of collections.

Probably the strongest improvement in MOF 2.0 is the possibility to define union-, subset-, redefinition-releationships between properties and hence association ends. In figure 2, the property *subdirectory* is defined as *subset* of *containedElement*. So only *containedElements* of type *Directory* can be a *subDirectory*. But without further constraints, there might be contained *Directories* that are not considered *subDirectories*. A *redefinition* relation allows to express stronger restrictions. The property *providedFile* redefines *file* and hence a *IncludeFolder* may only contain *files*, i.e. *providedFiles* of Type *HeaderFile*.

Having discussed the new concepts in MOF 2.0, we should mention that some features of UML 1.x do not exist in MOF 2.0. The most important features are stereotypes and qualifiers, but in our opinion, these are not essential for the purpose of meta modeling. The lack of stereotypes is barely a problem, as Fujaba only interpretes stereotypes to distinguish classes, interfaces, data types, references and Java Beans. In MOF 2.0, there are dedicated meta classes for most of these stereotypes, i.e. classes, data types, primitive types, enumerations and element imports. Qualifiers determine keys to quickly retrieve association ends from the collections implementing them. As meta modelling is about defining modelling languages, the name of a model element can be assumed to serve as qualifier where needed. So explicit qualifiers are not needed for meta modelling. Obviously, every known MOF or UML meta model has been defined without using qualifiers. Even if these meta models are far from perfect, most issues result from lacking constraints rather than missing qualifiers.

4. ADAPTING THE SDM EDITOR

Based on the internal meta model which is introduced in Fujaba 5, either UML or MOF can be used to define schemata for graph transformations. As we have seen in section 3, MOF provides some extensions that must be integrated in the graph transformation concept. In this section, we discuss how MOF 2.0 as schema language influences graph transformations in Fujaba.

Graph transformations in Fujaba, also referred to as Story Driven Modelling (SDM), require only very few concepts. On the one hand, activities, transitions and related guards allow to define the control flow of complex graph transformations. Choosing MOF 2.0 as schema language has no impact on these concepts at all. On the other hand, simple graph transformations are visually defined using a variation of collaboration diagrams. They consist of objects, links and multi-links, which are slightly affected by changing the schema language.

First of all, the advanced package concepts allow a better means to find available types for objects, associations and attributes than the existing solution with diagrams and views does. Namespaces and visibilities are taken into account and less name clashes occur, which result in nasty effects in the existing Fujaba implementation. For instance, a package merge as used in 2, effectively produces implicit generalization relationships between classes with identical names in both packages. When editing Story Diagrams, these additional generalization relationships must be considered during the creation of links, i.e. when the choice of possible edge types is determined.

The different kinds of inter-class relationships (associations and (mutual) references) in MOF 2.0 do not have impact on links in the first place. But when taking generalization, union, subsetting and redefinition into account, there are some affects on links.

The meta model implementation automatically maintains derived links which result from subsetting or redefintion. For instance, if an association end is marked as *subset* of another more general association end, changes to the subsetting end are automatically propagated to the subsetted end due to our meta model implementation. Considering the example in fig. 2, after creating a *subdirectory* link between two *Directories*, the *containedElement* link between *Directory* and *Element* does automatically exist as well and can be queried accordingly.

Next, if a general association end is marked as *union*, the association may not be directly modified by graph transformation rules. However, propagation from special association ends is still effective. Hence, the association becomes "abstract" so to speak. Unlike *redefinition* of association ends which will be discussed in section 5, the restrictions of *union* association ends can be analyzed statically.

In Fujaba, links in the set of Edges E have approximately

the following form [13]:

$$E \subset N \times EL \times I \times Q \times N \quad (1)$$

The first element denotes the source node, the second the edge label, the third the index (applies only for ordered associations), the fourth the qualifier (applies only for qualified associations), and the last the target node. Edge labels corresond to associations in the class diagram. Associations can be either plain, ordered or sorted, although UML like MOF 2.0 assigns this feature to association ends instead. However, MOF 2.0, does not feature sorted association ends.

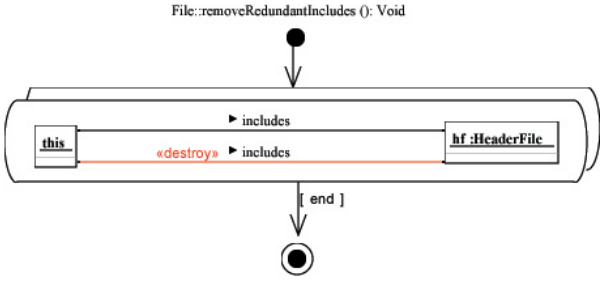

Figure 3: Remove redundant includes

Figure 3 shows an example of how non-unique association ends can be used to define a useful graph transformation for the schema in figure 2. If a *File* includes a *HeaderFile* more than once, the redundant includes should be deleted. Currently, Fujuba does not support these kinds of rules. The link is matched only once and then deleted. As a result, all inlude links will be removed.

In the case of ordered, non-unique association ends (sequences), multi-links might ostensibly allow to distinguish both links using multilinks. But a closer look to the generated source code in combination with the implementation of the related collection class reveils that in the general case, the multi-link cannot be matched.

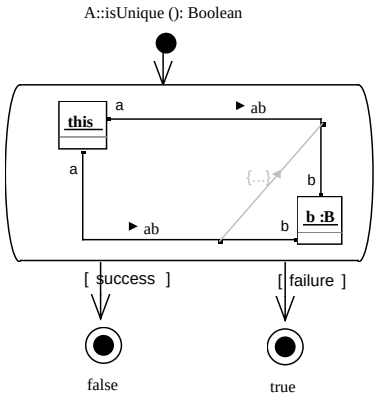

Figure 4: Detect non-unique links

Consider the example in fig. 4, and the resulting generated, but slightly condensed source code in fig. 5. The rule tries to detect two different links of the same *ordered* association between the same objects *a* and *b*. According to [13], p. 103, this should be possible: "..., in case of sorted or qualified associations there may exist multiple edges with the same label connecting nodes n1 and n2 as log as they carry different values for their index i or their qualifier q." We believe, that "sorted" in fact means "ordered", has sorted association ends are implemented using FTreeSets rather than FLinkedLists and hence no indices exist.

```
public boolean isUnique()
{
  boolean fujaba__Success = false ;
  B b = null ;
  Iterator iterB = null ;

  try {
    fujaba__Success = false ;

    //  multilink bind b : B iMultiLinkSearchNormTop
    iterB = this.iteratorOfB() ;
    while (!fujaba__Success && iterB.hasNext()) {
      try {
        b = (B) iterB.next () ;

        // check To-Many-Link 'b' between this and b
        JavaSDM.ensure (this.hasInB(b)) ;

        //  check multilink b to b
        JavaSDM.ensure (this.isBeforeOfB (b, b)) ;

        fujaba__Success = true ;
        //  iMultiLinkSearchNormBottom
      }
      catch ( JavaSDMException e ) {
      } // try catch
    } // while OOVariable[B,OOVariableType[12],]
  }
  catch ( JavaSDMException e ) {
    fujaba__Success = false ;
  }

  if ( fujaba__Success ) {
    return (false) ;
  }
  else {
    return (true) ;
  }
}
```

Figure 5: Source code for fig. 4

The crucial part is the call `A.isBeforeB(b,b)`. Tracing the call, it results in another call of `FLinkedList.isBefore(b,b)` (cf. fig 6), which obviously results in `false`, if `left` and `right` are identical as in this case, as `indexOf(b)` always finds the *first* occurence of `b`. The rule would probably work, if a method like `nextIndexAfter(b,b)` would be invoked for the second argument.

```
//returns true if left object is before right object
public boolean isBefore (Object left, Object right)
{
   return  (indexOf (left) < indexOf (right));
}
```

Figure 6: Source code for FLinkedList.isBefore(l,r)

As a result, association ends in Fujaba must always be unique with respect to the current implementation. Even the exception of ordered associations mentioned in [13] seems not to be implemented correctly. So dealing with non-uniqueness

requires modifications to Fujaba's graph model, although the precise impact is still under investigation. In case of ordered, non-unique association ends, this only seems to be a matter of improving the implementation of access methods. In case of unordered, non-unique association ends however, the formal definition of graph model would be affected. This does not necessarily mean to give edges an identity, but we would at least have to count plain links of the same edge type between identical nodes (LC), like

$$E \subset N \times EL \times I \times Q \times \mathbf{LC} \times N \quad (2)$$

The semantics of the rule in fig. 3 would be, that there is at most one tuple for every pair of instances of $File$ and $HeaderFile$, with a number c on the last but one position, representing the number of links. The rule matches, if such a tuple exists for $this$ and hf and $c > 1$. After the transformation, the tuple is modified and $c' = c - 1$[6].

5. BRINGING SDM AND JMI TOGETHER

The Java Metadata Interface (JMI) [3] is a standardized mapping of MOF compliant meta models onto Java. As the Java representation of MOF it specifies the generation of tailored interfaces for the creation and access of meta data as well as a set of reflective interfaces for a unified discovery of meta data. Furthermore, a JMI compliant meta model has to provide XMI [11] import and export functionality to facilitate an easy exchange of meta data. JMI compliant meta models can be implemented through various strategies and technologies. An implementation is compliant to the standard as long as the interfaces are met. The standardized interfaces facilitate an easy exchange and adaptation of meta models which make JMI a beneficial standard.

Thus, for the purpose of MOF based graph transformation JMI is the appropriate choice. Currently, MOFLON is able to generate an JMI compliant implementation for MOF 2.0 compliant (meta) models. Future versions will be able to complete that static implementation with the code which is generated by Fujaba SDM. The static part of the generated meta models will be generated by MOFLON's code generator and the dynamic parts with the graph rewriting engine of Fujaba. The current version of Fujaba uses proprietary Java interfaces for the static model representation (e. g. for multi valued association ends as listed in Table 1) which are used by Fujaba's graph tranformation. An utilization of Fujaba's graph transformation engine for the needs of MOF affords first of all a mapping between the interfaces generated by Fujaba and the interfaces demanded by JMI.

5.1 Mapping of Fujaba's interfaces on JMI

The major differences[7] between JMI and the code generated by Fujaba consist in the different handling of packages, class and association instances. These are differences on a large scale, but with just a limited impact on Fujaba's code generation as the static structure of the code will be generated by MOFLON. The most relevant parts concerning the dynamic implementation are the instantiation and deletion of class and association instances as well as the possibilities to navigate on the instance level.

Table 1 gives an exemplary overview how the mapping for the relevant parts might look like. For instance, JMI demands a more complicated class instantiation based on proxy classes representing the meta class. From the view of a code generator, the instantiation by calling a factory method is just syntactically overhead compared to the direct instantion using the new operator. The mapping for the deletion of instances and for single valued association ends is even more simple since the methods in both interface variants are nearly or even exactly the same. Multi-valued association ends are treated in JMI by making use of Java's Collection interface. Thus association ends are treated like ordinary attributes.

This feature reduces the interface to one single method but provides even more functionality through the Collection interface than the expanded interface generated by Fujaba does. There are other implementation concepts [6] that also reduce the interface to a single method. The main difference is that in [6] the links are stored in generic association end objects by using Java reflection for the creation of the backlink. In contrast, MOFLON generates a class for each association which centrally maintains the backlinks by using a listener concept. Due to the centralized storage the usage of Java reflection is not necessary. Nevertheless, the mapping still is no problem at all since each method of Fujaba's interface has a counterpart in the Collection interface. In the case of an ordered multi-valued association end[8] Fujaba generates convenience methods that do not have a counterpart in the List interface which JMI uses for ordered association ends. Those are methods that operate relative to the position of another linked object (e.g. addBeforeOfX). They will have to be replaced by an additional workaround of that parts of the implementation which use the non-mappable methods.

Considering navigability, some further aspects have to be taken into account. In terms of Fujaba, every association in MOFLON is navigable. The difference is, that navigable associations in MOFLON can be accessed as described in Table 1, whereas non-navigable associations can only be accessed by using the class representing the association. As a consequence, the call for accessing associations is expanded in such a way that the class representing the association has to be fetched from the package extent. After that, the handling of association instances is the same as for navigable associations. In general the interface mapping does not cause major problems at all.

5.2 Code generation for MOF 2.0

There are five major aspects that have to be considered as already mentioned in section 3. The package merge as one of the most significant features of MOF 2.0 demands no change in Fujaba's code generation subsystem for graph transfor-

[6] In the practice, this association should be ordered, as it matters which include has to be removed. But we decided not to introduce another example to discuss multi-relations here.

[7] The following comparison relates to the tailored (typed) interfaces of JMI because the reflective interfaces can easily be mapped on the tailored interfaces.

[8] The listing of that case is omitted.

	Fujaba	JMI
	new *Class*()	*package*.get*Class*().create()
	removeYou()	*refDelete()*
0..1	get$X$() set$X$(..)	get$X$() set$X$(..)
0..n	addToX(..) removeAllFrom$X$() hasInX(..) iteratorOfX() removeFrom$X$(..) sizeOfX()	get$X$(). *java.util.Collection* add(..) addAll(..) clear() contains(..) containsAll(..) *equals(..)* *hashCode()* isEmpty() iterator() remove(..) removeAll(..) retainAll(..) size(..) toArray() toArray(..)

Table 1: Comparison of code generated by Fujaba and JMI

mation since both kinds of package merge can be unfolded into regular model instances. After the process of unfolding the model instance is free of any package merges. A preprocessor which is part of the MOFLON/Compiler will execute this task directly before code generation. Thus the graph transformation will not even notice the existence of a package merge. After the code generation all changes in the meta model instance will be rolled back to prevent changes in the model due to the process of code generation.

Beside the package merge the subsetting and redefinition of association ends are major differences between Fujaba and MOF 2.0. Subsetting and redefinition are specified on the instance level and therefore potentially relevant for graph transformation. The subsetting of association ends causes instances of the subsetting association to be part of the extent of the subsetted association. Thus, the dynamic semantics of an ordinary subsetting do not cause any kind of restriction. The combination with union respectively with derivation in general at least demands a static analysis, which we have already discussed in section 3.

By definition, the redefinition of association ends causes the redefining and the redefined association end to share exactly the same set of links. Due to this constraint the instantiation of the redefined association is restricted. That restriction demands a runtime analysis to prevent forbidden instantiations as we will demonstrate using the *CProject* example. In Fig. 2 the redefinition of the association end *file* by the association end *providedFile* prevents an instance of association *has* between an instance of class *IncludeFolder* and an instance of class *File* since the collections of the association end *file* and *providingFile* for a given instance of *IncludeFolder* need to be the same. Thus, a rule as depicted in Fig. 7 may not be executed in the case that the rule is matching an instance of class *IncludeFolder* with an *element* which is not of type *HeaderFile*, whereas an execution in any other case does not cause any problems. Note, that the general rule defined for the package *FileSystem* is interfered by the more special package *CProject*. But such a case can only be detected during runtime and not by static analysis. In our opinion, wrong usage of redefined association ends is a specification error and should result in a runtime exception.

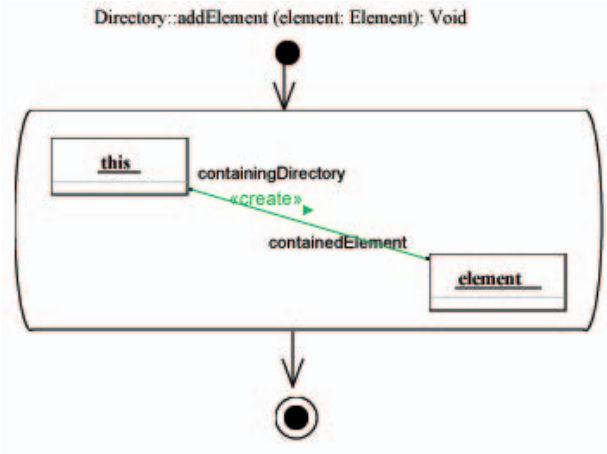

Figure 7: A general rule affected by redefinition

Other features whose compliance can only be checked at runtime are the upper and lower bounds of association ends and the composite constraint. Currently, Fujaba ignores the dynamic semantics of those features, whereas MOFLON at least tries to keep models consistent by deleting those objects whose lower bounds and composite constraints are not met. Such a solution guarantees consistent meta model instances in cases without an automated transformation. During automated transformation temporary inconsistent meta model instances might be desirable. Therefore, in future versions a sophisticated constraint evaluation mechanism combined with repair actions comparable to similar concepts in PROGRES [8] will control the compliance of any kind of constraint. This mechanism will keep the graph rewriting engine free of any contact with constraints.

A last aspect that has to be considered is the different usage of MOFLON's kinds of associations. Since both associations and (mutual) references in MOFLON are mapped on associations in Fujaba, stereotypes have to be used to control Fujaba's code generation regarding the usage of different code variants for associations. The code generation needs to be flexible enough to take this aspect into account.

6. SUMMARY

In this paper, we have discussed the implications of using MOF 2.0 as schema language for Fujaba graph transformations. This adaption involves some modifications of different parts of Fujaba which are summed up in Table 2. As already indicated in [13], the introduction of unique association ends requires modifications of the graph model. Features like union require additional statical analysis rules, whereas others (e.g. redefinition) demand an analysis during runtime. The remaining features are already covered by our implementation of the MOF meta model.

	SDM Editor	SDM Compiler	MOF Compiler
Packages	static analysis	√	preprocesing
Attributes union	static analysis	√	√
subset	√	√	runtime propagation
redefines	√	(√) runtime exeception handler	runtime propagation
unique	√	?	√
ordered	√	?	√
Navigability	√	parameterized code generator templates	√
Composition	flexible constraint checking and repair action concept		
Multiplicity			

Table 2: Impacts of the adaption of MOF 2.0

Apart from the ongoing effort to integrate MOF 2.0 in the Fujaba 5 main branch, the next steps from our point of view are the integration of OCL constraints with repair actions and triple graph grammars [5]. Besides, we have to put some more effort in the correct implementation of namespaces defined by packages.

7. REFERENCES

[1] C. Amelunxen. Building a MOF 2.0 Editor as Plugin for FUJABA. In A. Schürr and A. Zündorf, editors, *FUJABA Days 2004*, volume Technical Report tr-ri-04-253, pages 43–47. Universität Paderborn, 2004.

[2] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. Adapting FUJABA for Building a Meta Modelling Framework. In H. Giese and A. Zündorf, editors, *FUJABA Days 2003*, number tr-ri-04-247 in Reihe Informatik, pages 29–34. Universität Paderborn, 2003. Technical Report.

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

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

[5] A. Königs and A. Schürr. Tool integration with triple graph grammars - a survey. *Electronic Notes in Theoretical Computer Science*, 2005. Submitted for publication.

[6] T. Maier and A. Zündorf. Yet Another Association Implementation. In A. Schürr and A. Zündorf, editors, *FUJABA Days 2004*, volume Technical Report tr-ri-04-253, pages 67–72. Universität Paderborn, 2004.

[7] M. Matula. *NetBeans Metadata Repository*. SUN Microsystems, March 2003.

[8] M. Münch, A. Schürr, and A. Winter. Integrity constraints in the multi-paradigm language progres. In *Proc. 6th International Workshop on Theory and Application of Graph Transformations TAGT, Paderborn*, volume 1764 of *Lecture Notes in Computer Science*, pages 338–351, Heidelberg, 2000. Springer Verlag.

[9] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification*, March 2003. ptc/03-10-04.

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

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

[12] T. Rötschke. Adding Pluggable Meta Models to FUJABA. In A. Schürr and A. Zündorf, editors, *FUJABA Days 2004*, volume Technical Report tr-ri-04-253, pages 57–62. Universität Paderborn, 2004.

[13] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. Habilitation Thesis.

Detection of Incomplete Patterns Using FUJABA Principles

Sven Wenzel
Tampere University of Technology, Finland
University of Dortmund, Germany
email@svenwenzel.com

ABSTRACT

An approach for the detection of structural patterns in UML class diagrams is presented. It picks up some principles of the reverse engineering component of FUJABA, such as a hierarchical pattern definition and an alternating bottom-up/top-down analysis. Furthermore it uses a fuzzy-like evaluation mechanism so that it is able to recognize not only entire patterns but also incomplete instances.
The knowledge about incomplete instances, which obviously occur rather often during the developing process, assists developers not only while maintaining or reverse engineering existing software but already while designing and implementing new software.

1. INTRODUCTION

Since software systems become larger and more complex, the task of understanding while developing and especially while maintaining software becomes more and more difficult. Therefore the use of patterns has become a helpful methodology to develop software in a more structured and understandable way.

In general, a pattern is a scheme that consists of three main parts: a context, a problem, and a solution. They discuss a particular recurring problem which arises in a particular situation – the context. Furthermore they offer a proven solution to this problem.

Several pattern families focus on different aspects of software development and take place in the different stages of the development process. The main interest within the design and implementation phases is directed towards those patterns that focus on problems of software design. These are behavioral patterns that focus on the run-time behavior of software elements and structural patterns that concentrate on the structural arrangements of software elements. Both types can be separated into more abstract and more concrete problems.

Design patterns [1] are more abstract and focus on problems in object-oriented software in general. For example, the *Composite* pattern describes how to compose several objects into a part-whole hierarchy with a uniform interface. Specialization patterns [2] do not discuss general project-independent problems, but focus on more specific topics of concrete projects. They describe, for example, how to extend a particular framework and support users in this process.

It is well-known that the knowledge about patterns used in a software helps the developer to get a better understanding of it. Therefore, FUJABA [3] and other tools (see Section 7, Related Work) offer mechanisms to detect pattern instances in given software. Especially in maintenance and reverse engineering the occurrence of the solution parts of particular patterns help the developer to understand the original problems.

But the knowledge about used patterns – especially with regard to structural patterns in general – assists the developer already while designing or implementing new software. As in maintenance and reverse engineering, the reason is a better understanding, but the circumstances are quite different. Since the software is currently in development, the likelihood of partly instantiated patterns is rather high. However, the knowledge about those incomplete patterns helps the developer and should not be neglected.

An example is the use of specialization patterns guiding a developer while extending a framework. The patterns define the classes that have to be created, the interfaces that have to be implemented, or the operations that have to be overwritten, etc. Furthermore, architectural rules may be defined as a pattern to enforce structural properties of the developed software. The developer would like to check if her design satisfies these architectural rules during development and she also wants to have information about the work progress, in other words, the tasks that have to be done.

Consequently, an approach for the detection of structural patterns should be able to recognize these incomplete instances as well as entire ones. A possible realization of the detection of incomplete patterns based on the Unified Modeling Language (UML) [4] is generally presented in [5] and is also discussed on the next pages. It picks up some fundamental ideas of the FUJABA approach, such as a hierarchical pattern definition and an alternating bottom-up/top-down analysis. Furthermore it uses a fuzzy-like evaluation mechanism so that the introduced approach is able to recognize not only entire patterns, but also incomplete instances.

Due to its parallels to FUJABA, this paper focuses on the differences between this approach and the one underlying FUJABA. The following Section 2 introduces the FUJABA approach itself, while Section 3 discusses the characteristics of incomplete pattern instances and the limits of FUJABA. Sections 4 and 5 present an improved model for pattern

definitions and the new recognition approach itself. The implementation of this approach is discussed in Section 6. Finally, Section 7 summarizes related approaches for pattern detection and Section 8 summarizes current and future work.

2. DETECTION WITH FUJABA

FUJABA stands for "From UML to Java and back again" and provides a round-trip engineering tool based on UML. It allows developers to design and implement software both visually and code-based. UML class and behavior diagrams allow the formal definition of software design and run-time behavior, while the source code is generated automatically from the diagrams. In turn, changes in the source code cause changes in the diagrams to keep the whole system consistent.

The reverse engineering component of FUJABA furthermore offers a powerful tool to detect design patterns automatically and gives thereby developers a better understanding of the software they reverse-engineer with the FUJABA as motivated in the introduction.

The detection mechanism itself [6] works on the abstract syntax graph (ASG) of the investigated software, which is a formal software representation eliminating most of the syntactical variants and formatting problems. The mechanism is based on graph grammars working on the ASG and the patterns to be detected are defined by graph transformation rules.

Each rule transforms a particular graph structure, i.e. a pattern or a subpattern, and annotates it with an additional node to indicate the found instance and additional edges to indicate the participants of this instance. Accordingly, the patterns are defined as compositions of other patterns or structures.

As an example Figure 1 shows the transformation rule for the generalization structure in FUJABA. The three nodes and two edges on the right side represent a small extract of the ASG that shows the inheritance between two classes. The transformation rule annotates this structure as a generalization and indicates the super class and the sub class as participants. Furthermore, it is possible to add *optional nodes* to the rule. These optional participants of a pattern are annotated, too, if they are found, but they are not necessary for the pattern instance itself.

The annotation node is additionally tagged with two percentage values. The first value is the reliability expressing how sure the pattern instance can be assumed, if the annotated participants are found. The second value provides a threshold defining the minimal necessary reliability of all subpatterns to mark the found pattern as an instance.

Once this generalization is found and annotated in the ASG, larger patterns or structures using it can be detected. Figure 2 for example shows the rule for the *Strategy* design pattern. It consists, among others, of the generalization structure.

All patterns and structures are organized in a graph that shows the compositions of patterns and substructures and builds a dependency hierarchy between them. Based on this hierarchy the detection algorithm is able to find pattern instances, proceeding in an alternating bottom-up/top-down analysis that annotates the ASG stepwise with new elements.

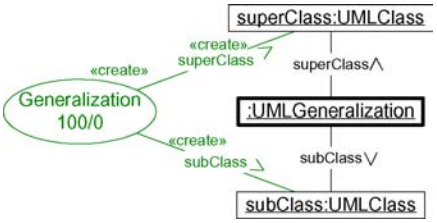

Figure 1: The generalization structure in FUJABA (taken from [7]). The oval node indicates the generalization structure in the ASG. The edges refer to the participants of the generalization.

Figure 2: The Strategy pattern in FUJABA (taken from [7]).

In the beginning all transformation rules of the lowest level in the hierarchy are applied to the ASG, assuming that the circumstances permit it. Then the algorithm starts in a bottom-up strategy. It analyzes the rules applied to the ASG, and also tries to apply the transformation rules of the above patterns which depend on these rules. In the case that a higher rule cannot be applied because one or more prerequisites are missing, the algorithm switches to a top-down analysis to apply the necessary rules. If these rules are applicable, the algorithm applies them and switches back to the bottom-up mode afterwards to proceed normally. In case the necessary rules cannot be applied, the prerequisites of the higher rule are missing. Thus the rule is not applicable and the algorithm continues with the other rules in bottom-up strategy.

3. INCOMPLETE PATTERNS

The detection algorithm of FUJABA does not apply a transformation rule, if some of its prerequisites are missing, as it is usual for a grammar-based approach. Thus, unsatisfiable pattern candidates are eliminated immediately.

However, sometimes the pattern is formally not there, but it is supposed to be there – either because parts of the pattern are missing or because some properties are wrong. If the strategy class is not defined as abstract, for example, the whole pattern would not be detected, even though it is actually there.

Of course, if we talk about patterns that are described in a formal and precise way, the instances of those patterns have

to be precise as well. However, the absence of some conditions does not necessarily mean a nonexistence of the pattern, but it can mean the incompleteness of the pattern instance. Especially while developing new software, this kind of incomplete instances occurs rather often. To refer to the last example, the developer had just not marked the strategy class as abstract yet.

According to the fact that pattern instances might exist even if some of their conditions are not totally satisfied or some of their roles are missing, a detection mechanism should be able to deal also with these incomplete instances. The first step in that direction is already given by the optional nodes of FUJABA. In this case each pattern is defined several times – each time with different nodes marked as optional. This procedure might end up in a cumbersome task for the user and provides furthermore insufficient information about the quality of found instances. Therefore, the following sections present a new approach that provides a mechanism to deal with incomplete instances.

4. ROLE-BASED PATTERN DEFINITION

Since the new approach focuses on the search of those incomplete instances that cannot be found with FUJABA, it avoids the use of grammar-based mechanisms. Thus, the patterns are not expressed as transformation rules, but a pattern definition is used that is applicable to the detection of incomplete patterns, so that the absence of some parts or properties can be easily reflected to the pattern definition.

Detached from the pattern type, the solution part of a structural pattern defines an arrangement of software elements to solve a particular problem. However, the problem itself is not of interest here and the arrangement of software elements is rather a template than a combination of concrete software elements. These template elements are called *roles*. They are placeholders which can be taken from concrete elements in the instance of the pattern. Each role has a type (e.g. classifier or association) to determine the kind of software elements that can act as the role. Since software elements allow the nesting of other elements, each role may contain several subroles representing nested elements as the patterns in FUJABA allow the nesting of subpatterns.

However, the existence of roles and their nesting relations in between is not sufficient to express complex arrangements of software elements, so that roles can be enhanced by constraints. These constraints are given in the Object Constraint Language (OCL) [4, Chap. 6] and enforce certain properties of the concrete elements acting as the role. They define, for example, visibility or stereotype properties. Furthermore they may refer to other roles to express particular relations like inheritance or parameter types.

By default every role has to be played exactly once in a pattern instance, but it is possible to define multiplicities to give limitations for the amount of elements acting as a role in a pattern instance. The multiplicity provides a lower and an upper range as it is done for association ends in UML. A lower range of zero makes a role optional and an infinite upper range allows as many elements acting the role as possible.

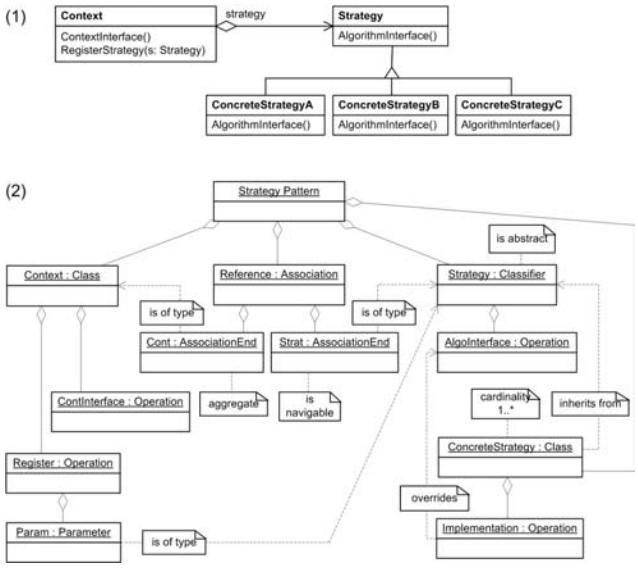

Figure 3: The Strategy pattern pictured as (1) a UML class diagram (from [1]) and (2) as a role-based pattern definition. In this example the pattern is extended by an operation for registering a strategy.

An example for the role-based representation is shown for the *Strategy* pattern in Figure 3. Each element of the UML class diagram describing the pattern like in [1] is translated to a role. The type of the UML element determines the type of the role. Child elements (e.g. parameters of operations) become subroles and properties of elements (e.g. abstraction or inheritance) are replaced by constraints for the corresponding role.

The graphical notation for the pattern definition used in this article is a UML object diagram extended by some features of UML class diagrams. Object nodes represent roles – labeled with the name and the type of the role, separated by a colon. Aggregations express containments of subroles and constraints are represented by notation elements. For dependencies between roles dashed arrows are used. Note that the constraints are given informally to simplify the diagram. Usually they are expressed in OCL like for example `context:TypedElement inv:self.type=...` instead of "*is of type*".

5. THE NEW DETECTION APPROACH

The previously presented pattern definition has its advantages in its notation and in its structure that can be transformed to a hierarchy as the one in the FUJABA approach.

The notation based on a UML object diagram describes exactly what to search for. The developer wants to find an object of the type *class* that acts as the *context* role (cf. Figure 3). Furthermore this class object should contain an operation object that acts as the *register* operation. This operation object again should contain a parameter and so on.

This situation of picking elements for particular roles can be compared to a casting for a theater play. Result of the

search is a set of mappings between particular roles and the elements acting as these roles, whereas the elements are called *candidates* and the whole set is called a *cast*. The single mapping between one role and a candidate for this role is called a *binding*; it binds a candidate to its role.

Each binding is associated with a quality value that expresses how well the candidate acts as the role. As in the world of theater there will exist several candidates for a particular role or one candidate for several roles, however, in each case the role is treated with a different quality. Therefore the algorithm does not bind candidates to roles by simple *yes-or-no* decisions, but rather uses assignments with intermediate values similar to those usual in fuzzy-logic.

These intermediate values are expressed in the quality of a binding. The quality values range between 0% and 100%. The value is zero, if the candidate cannot act as the role at all; 100% means that the candidate satisfies all of its requirements. The value itself is calculated by the constraints and the subroles.

Generally it is possible to create every binding right from the start, because if there is a role of type *class*, every class could be a candidate for this role. Even if all constraints and subroles are unsatisfied, it is still a class and consequently a candidate for this role. This technique will obviously end up in an unmanageable set of bindings and has to be more organized.

Here the second advantage of the role-based pattern definition comes into play. The roles can easily be classified into a dependency hierarchy like the one in the FUJABA approach to improve the binding process. This classification is the first phase of the detection algorithm. Then the detection proceeds with a bottom-up/top-down analysis. Different from the FUJABA approach, the bottom-up and top-down strategies do not alternate several times during the detection of a pattern, but only once. Therefore, they can be regarded as the second and third phase. The bottom-up phase locates all candidates for each role and is hence comparable to the bottom-up phase of FUJABA that marks subpatterns by annotating them. The successive top-down phase selects the best candidates for the searched pattern. Similar to FUJABA it thereby validates the pattern instance.

5.1 Classifying Phase

The definition of the pattern is a graph of roles connected by child or dependency relationships. The child relationships can also be considered dependencies because an element should act as a particular role only if also its children act as the corresponding subroles.

Thus, the graph can be sorted by those dependencies to organize the roles into different levels (see Figure 4). The lowest level contains all roles that have no dependencies to other roles. Every next level contains all roles that have dependencies only to roles from the lower levels. The highest level contains only one virtual role that represents the pattern itself. It depends on its children – the main roles of a pattern.

The generated hierarchy is the basis for the detection similar to the one in the FUJABA approach. If the hierarchy is processed level-by-level from the bottom to the top as it is done in the bottom-up phase, it is ensured that supplying roles are always checked before their clients.

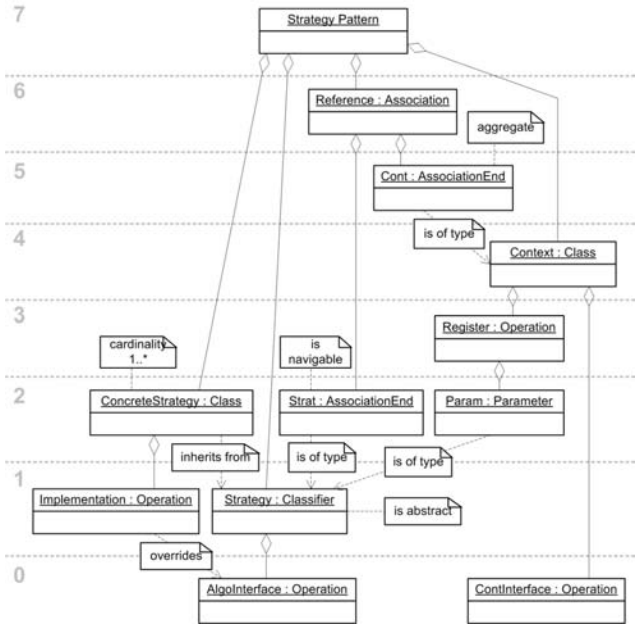

Figure 4: The ordered role graph of the Strategy pattern. The roles are arranged on different levels based on their dependencies which are either caused by constraints or parent-children relationships.

5.2 Bottom-Up Phase

The bottom-up phase searches the candidates for each role. It starts with the roles at the lowest level of the beforehand generated hierarchy, as they are independent from any other role.

In the beginning the algorithm locates all elements from the model, which can act as the particular role by their type and creates a temporary binding. Different from the procedure in FUJABA, the binding process does not pay any attention to the constraints or prerequisites here. Thus, in our example every operation of the entire model could act as the role *AlgoInterface* (cf. Figure 4) and for each of those operations a temporary binding is created. Indeed, this procedure will cause a huge amount of bindings. If r roles are bound to n model elements, the number of bindings equals the variation of r out of n elements regarding the order and allowing repeats, which is $V_n^{(r)} = n^r$. However, this is an upper limit, since the problem is divided into several smaller variations, as for example *class* roles are only bound to *classes*. The method might still be inefficient for large software models, but two arguments legitimate this lack of efficiency. First, the approach is supposed to be applied on UML models during the design phase. Thereby a natural limitation of elements can be assumed as the models are developed by human beings. Second, the developer is especially interested in incomplete instances and thereby it is necessary to investigate each part of the model in detail. Too fast an

elimination of candidates could cause an overlooking of possible instances.

Once a role is temporarily bound to elements, the constraints assigned to the role are evaluated for the bound element, because the element type is not the one and only criterion for bindings. In contrast to the idea in FUJABA, the constraints do not determine the applicability of bindings, but rather their quality, because also partially satisfied bindings are of interest. Thus, the evaluation of a constraint results in a value between 0% and 100% expressing how well the constraint is satisfied. The value of a simple constraint, e.g., *"is abstract"*, is 100% in case of satisfaction, 0% otherwise. A more complex constraint may also return intermediate values if it is satisfied partially. An example is a naming constraint where only the prefix of the name fulfills the requirements.

If a role of the lowest level has no constraints at all, there are no constraints to assess the binding and it is taken as fully satisfied. Otherwise the quality of the binding is reflected by the average quality of the constraints. Thus the mean of all constraint evaluation results is calculated and taken as the quality of the binding:

$$q(b) = \frac{1}{|C|} \cdot \sum_{i=1}^{|C|} eval(C_i) \quad \text{if } |C| > 0 \qquad (1)$$

whereas $q(b)$ is the calculated quality of a binding b, C is the set of constraints, and $eval(C_i)$ is the evaluated quality of the constraint with index i.

After all the roles of the lowest level have been bound and their constraints evaluated, the algorithm continues with the roles from the next higher level. In contrast to the bottom-up strategy of FUJABA, the bottom-up phase of this approach does not only process the clients of the previously processed role, but all roles of the next level.

Constraints of roles from all levels higher than zero depend on other roles. Thus, the returned value of the constraint evaluation is dependent on the quality of the binding referred to by the constraint. An example is the constraint evaluation of *Implementation* (cf. Figure 4): The *overrides* constraint defines that the operation bound to *Implementation* has to override an operation bound to *AlgoInterface*. The evaluation returns 0%, if the operation found does not override another operation at all or the overridden operation is not bound to *AlgoInterface*. Otherwise it returns the quality of the binding between the overridden operation and the role *AlgoInterface* which should be overridden here. Since this binding has a particular quality, the constraint is satisfied by this quality.

Furthermore, the roles from all levels higher than zero may contain also subroles. Each of those subroles, S_i, can be interpreted as a constraint *"has a child of type S_i"*. Thereby, missing subroles do not eliminate a candidate, but they reduce its quality. Consequently, the calculation of the quality is extended by those subroles S, which are handled in the same manner as additional constraints.

$$q(b) = \frac{1}{|C|+|S|} \cdot \sum_{i=1}^{|C|} eval(C_i) + \frac{1}{|C|+|S|} \cdot \sum_{i=1}^{|S|} q(S_i) \qquad (2)$$

whereas S is the set of subroles of the role of the binding b (i.e. the currently analyzed) and $q(S_i)$ is the quality of the binding found for subrole S_i. The other variables and functions are equal to the ones in equation (1).

These subrole constraints are calculated like those constraints depending on other roles – here, the subroles. If no candidate for the subrole (i.e. a child element in the model) is found, the constraint is treated as 0%. Otherwise it has the quality of the binding of the candidate found.

The algorithm traverses upwards the dependency graph from level to level as described above and for each role new temporary bindings are created and evaluated. Unlike in FUJABA, the algorithm does not change to the top-down phase, since all roles are processed level-by-level and consequently the necessary suppliers for each binding are calculated beforehand.

All unsatisfied bindings (i.e. those with a quality of 0%) are deleted. The other bindings, which satisfy at least some constraints or subroles, are stored in a cast for the later top-down phase. Thereby, the threshold for keeping and deleting bindings can optionally be changed from zero to any other value. For example, to 50% to keep only bindings which are half satisfied. Choosing 100% would yield a detection of complete patterns.

The result of the bottom-up phase is a set of bindings from roles to different elements. Each binding has a particular quality that expresses how well the role is acted by its element.

5.3 Top-Down Phase

The detection algorithm switches to the top-down phase when the virtual role representing the pattern is reached (i.e. the role of the highest level). The cast generated so far contains a lot of bindings that are not necessarily part of a pattern instance, especially if the threshold was set to a small value or zero. These bindings are called *false bindings* and are filtered out in this phase of the algorithm.

Therefore the original pattern definition is now handled as a role tree by taking the child relationships as edges and the virtual role of the pattern as the root. Different from the dependency graph, the dependencies are not taken as edges, they just give the information how many clients a role has.

This role tree is now traversed downwards in a breadth-first search order. For each role the binding with the highest quality is selected to be kept in the final cast. In case of equal qualities of multiple bindings for a role it is checked which binding is supplier for other bindings. Bindings with more clients than others are preferred. Also the information about other roles acted by the element of the binding is analyzed. It might be that an element that acts as the current evaluated role is the only candidate for another role. Especially if an element is supposed to act as only one role at a

time, it has serious consequences for which role the element is taken.

During the entire procedure the child relationships of the actual elements in the investigated model are considered and thus only the children of a particular element in the model can act as the subroles of the role of that element. Furthermore the role multiplicities are considered in a way that roles with an upper range greater than one allow the selection of a suitable amount of bindings.

The top-down phase results in the cast that contains all bindings representing an instance of the searched pattern.

5.4 Several Pattern Instances

The usage of graph grammars in the FUJABA approach supports the detection of multiple pattern instances a priori, because the transformation rules can annotate the ASG at several locations, once for each pattern instance.

Consequently, this support for finding multiple pattern instances has to be added to the new algorithm. If taken as presented, the algorithm would just result in one pattern instance found. It would be the most satisfied one; however, the developer is interested in all instances.

Therefore, the bottom-up phase is accompanied by an assignment procedure that maps the found bindings to possible pattern instances with respect to the fact that one binding can occur in several overlapping patterns. For example, a model contains different *Contexts* that use the same *Strategy*.

Each newly found binding either belongs to a new pattern instance that has not been touched so far, or it belongs to one that has already other bindings assigned. Thus, each binding is assigned to a new pattern instance, or, if the binding has relationships to other bindings (i.e. children or dependencies), it is mapped to the same instances those related bindings belong to. If the role *AlgoInterface* for example is bound, the binding is assigned to a new pattern instance, because it has no dependencies. Whereas the binding of *Strategy* belongs to the same pattern instance as the binding of its child (i.e. the *AlgoInterface* operation) belongs to.

These assignments between bindings and pattern instances are later respected in the top-down phase. The phase is not performed only once, but for each possible pattern instance. The top-down analysis thereby consults only those bindings that belong to the currently investigated pattern instance.

Once again, a predefined threshold ensures that dispensable pattern instances are deleted. For example, a binding has been assigned to a pattern instance but it has never been referred to by another binding; obviously this instance containing one binding only is not very promising. The default value is set to 50%. Of course, a threshold of 100% would just keep complete instances.

With respect to the assignments between bindings and pattern instances, the algorithm results in a set of casts for different pattern instances. Each found pattern instance is annotated with a quality value, in the same way as the qualities of single bindings. Figure 5 shows an exemplary result of the new approach detecting the Strategy pattern in a class diagram. Two instances have been found; one with a quality of 100% and another one with a quality of 75%, since its role *ContInterface* is missing. Both instances share the same *Strategy*. A third possible instance has only a quality of 43% and does not reach the threshold, since the *Register* role is missing and the abstraction constraint is unsatisfied. Of course, there are also a lot of other instances containing one element only.

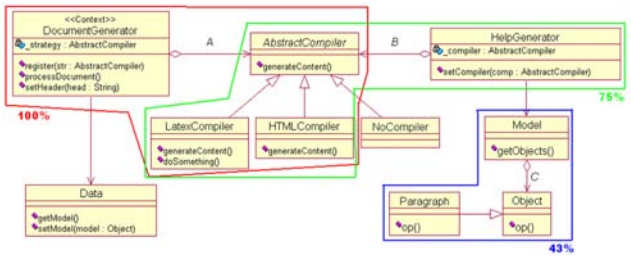

Figure 5: An exemplary result of the new approach detecting the Strategy pattern.

In contrast to FUJABA, the result of this approach is enriched with the information about incomplete pattern instances.

6. IMPLEMENTATION

The previously introduced approach has been implemented as a layered architecture to keep it as independent as possible from any particular semantics of the investigated models.

The core algorithm works just with semantic-less objects. The only assumption made is that the models on which it works have a compositional structure, so that the parent-children relations of the roles are reflected in the model. That way the bottom-up and top-down phases of the algorithm can work independently from the model by using an adapter encapsulating the model access.

Since the adapter hides the semantic-specific implementations, the algorithm can request the parent of an element regardless of the element being an operation nested by a class or a class nested by a package. The handling of constraints is also processed by this semantics adapter. Thereby it is possible to provide different types of constraints, checking for instance stereotypes in UML diagrams which is unpractical for Java code. Furthermore the usage of adapters supports different evaluation methods, so that for example the OCL interpreter working on class diagrams is differently implemented than the one working on syntax graphs.

As shown in Figure 6, the search engine itself takes just the pattern definition as an input. This definition consists of roles and constraints and is consequently independent from any specific pattern type. The roles themselves just know their relations to other roles and store the information on assigned constraints. Furthermore, they know the element type they can be bound to. Only the constraints are related to the model semantics and are therefore placed on the second level.

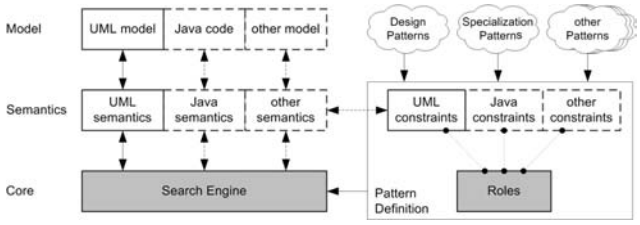

Figure 6: The introduced approach has been implemented as a layered architecture.

So far, the semantics have been implemented only for UML class diagrams. The *Eclipse UML2* project [8] provides the basis for representing the models and the implementation of the semantics layer works directly on these models. Most of the constraints have been realized on basis of OCL using the *Kent OCL Library* [9]. Simple boolean OCL expressions allow to check if a particular property is satisfied, e.g., `context NamedElement inv: visibility='public'`. Other OCL expressions can return an element of the model that is checked to be bound to a particular role, like `context Class inv: self.generalization.general= [Strategy]` to check inheritance. Naming and stereotype constraints use proprietary implementations that allow regular expressions.

A graphical user interface is not provided so far. Hence, the visualization of patterns and their instances in a class diagram is not given. The current implementation is rather a library providing the search mechanism to be integrated in other tools.

7. RELATED WORK

Obviously the FUJABA *Tool Suite RE* is the most related approach to detect patterns, as some basic ideas have been reused in the introduced approach, even if FUJABA focuses on detection of complete pattern instances. So far the FUJABA approach concentrates on structural patterns, but the support for behavioral patterns is already researched [10]. Among FUJABA, there exist a lot of other solutions noted for automated detection of design patterns while maintaining or reverse engineering software.

Heuzeroth et al. [11] present an approach based on predicate logic. It operates in two phases and is able to detect behavioral patterns as well. First, candidates for pattern instances are searched with the help of predicates in a static analysis. Second, the runtime behavior is analyzed to check if the expected behavior for the candidates is satisfied.

An approach based on software metrics is presented by Antoniol et al. [12]. It calculates class level metrics and compares them to a previously defined catalog of pattern metrics.

Keller et al. [13] deal with an intermediate representation of source code. It is held in a design repository that is based on a relational database and provides storage as well as querying of abstract design components that are in fact nothing else than pattern definitions.

Krämer and Prechelt [14] present an approach based on PROLOG. Both, design patterns and software designs, are expressed in PROLOG terms. The search for patterns is then done by the PROLOG engine.

An approach to detect design patterns with relational algebra is researched by Fronk and Berghammer [15]. Design patterns and structural information of a software system are expressed in relational terms. A relational calculator is used to solve the relational equations and recognizes the pattern instances.

8. CURRENT AND FUTURE WORK

The current work focuses on the integration of the search engine into *MADE* [16], an architecting tool that assists designers to define pattern-based architectures and developers to instantiate them. The knowledge of incomplete instances will assist the developer and allows the tool to process some tasks automatically.

As part of this integration it is aimed to allow a manual definition of bindings. Thus the users are able to assign elements to roles by themselves and the tool finds the related roles or reports their absence respectively. Furthermore, the integration of the new approach into the tool will allow empirical case studies to prove the usability in industrial-size software systems.

One future objective will be the extension to semantics of programming languages, such as Java. This would increase the possible field of application of this approach, especially in reverse-engineering. A realization by using the abstract syntax graph of the source code would yield in a solution that might be integrated into FUJABA.

However, the main future objective is continual improvement of the search engine itself. The possibility to assign single constraints with weights will allow a more precise pattern definition, as some roles or constraints are more important than other ones. Furthermore it is planned to add support for behavioral patterns. In case of handling them as well as structural patterns the possible field of application will be increased much more.

9. ACKNOWLEDGEMENTS

The approach has been developed as part of the author's diploma thesis, written in cooperation between the Tampere University of Technology, Finland, and the University of Dortmund, Germany. The thesis is funded financially by the Martin-Schmeißer-Stiftung of the University of Dortmund.

10. REFERENCES

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paaki, Antti Viljamaa, and Jukka Viljamaa. Annotating reusable software architectures with specialization patterns. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, pages 171–180, Amsterdam, August 2001.

[3] FUJABA Tool Suite Developer Team - University of Paderborn. FUJABA Tool Suite. `http://www.fujaba.de/`, June 2005.

[4] The Object Management Group. Unified Modeling Language Specification – version 1.5 (formal/03-03-01). Online at `http://www.omg.org/uml/`, March 2003.

[5] Sven Wenzel. Automatic detection of incomplete instances of structural patterns in UML class diagrams. In *Proc. of the 3rd Nordic Workshop on UML and Software Modeling (NWUML'05)*, Tampere, Finland, August 2005. [to appear].

[6] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proc. of the 24th ICSE'02*, pages 338–348, Orlando, Florida, USA, May 2002.

[7] Lothar Wendehals. Cliché- und Mustererkennung auf Basis von Generic Fuzzy Reasoning Nets. Diploma thesis, University of Paderborn, Germany, October 2001.

[8] Eclipse Foundation. Eclipse UML2 Project. `http://www.eclipse.org/uml2/`, June 2005.

[9] Dave Akehurst and Octavian Patrascoiu. Kent OCL Library. `http://www.cs.kent.ac.uk/projects/ocl/`, June 2004.

[10] Lothar Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proceedings of the ICSE'03 Workshop on Dynamic Analysis (WODA'03)*, Portland, USA, May 2003.

[11] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *Proc. of the 11th IWPC'03*, pages 94–103, Portland, Oregon, USA, May 2003.

[12] Giuliano Antoniol, Roberto Fiutem, and Luca Cristoforetti. Design pattern recovery in object-oriented software. In *Proc. of the 6th IWPC'98*, pages 153–160, Ischia, Italy, June 1998.

[13] Rudolf Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *Proc. of the 21st ICSE'99*, pages 226–235, Los Angeles, California, USA, May 1999.

[14] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proc. of the 3rd Working Conference on Reverse Engineering (WCRE'96)*, pages 208–215, Monterey, CA, USA, November 1996.

[15] Alexander Fronk and Rudolf Berghammer. Considering design problems in oo-software engineering with relations and relation-based tools. *Journal on Relational Methods in Computer Science (JoRMiCS)*, 1:73–92, Dezember 2004.

[16] Imed Hammouda, Juha Hautamäki, Mika Pussinen, and Kai Koskimies. Managing variability using heterogeneous feature variation patterns. In *Proc. of FASE'05*, pages 145–159, Edinburgh, UK, March 2005.

Calculation and Visualization of Software Product Metrics

Matthias Meyer*
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
mm@uni-paderborn.de

Jörg Niere
Software Engineering Group
Department of Computer Science
University of Siegen
Hölderlinstr. 3
57068 Siegen, Germany
joerg.niere@uni-siegen.de

ABSTRACT

The paper presents a further step of the Fujaba Tool Suite RE to support coarse-grained analyses based on metrics and especially polymetric views. Polymetric views are graphical representations of certain metric combinations. Following an interactive reverse engineering approach, polymetric views can be created on demand. The reverse engineer is able to define new polymetric view descriptions and create new views afterwards.

1. INTRODUCTION

Software product metrics are one opportunity to perform coarse-grained analyses. Metrics such as lines of code, number of attributes or methods of a class, lack of cohesion or depth of inheritance hierarchies ([3, 4, 5, 7]) allow for producing quantitative analysis results of a software system. A combination of different metrics allows to draw conclusions such as problematic or high influencing system parts. To overcome the flood of numbers produced by the metrics, Lanza proposes in [6] a graphical representation of the metric combinations. So-called polymetric views are an ideal means to get a first impression of a system.

The Fujaba Tool Suite RE is a collection of reverse engineering tools based on the Fujaba Tool Suite [10] and several Fujaba plug-ins. The Fujaba Tool Suite RE allows for parsing Java source code into an Abstract Syntax Graph (ASG) representation, which serves as central repository to all further analyses. Currently the Tool Suite RE consists of static and dynamic analysis techniques to recognize implementations of patterns [8], such as design patterns or antipatterns [1]. The techniques allow for performing a fine-grained analysis of a system. Coarse-grained analyses are also possible, but produce too many uncertain results.

In order to support also fast and reliable coarse-grained analyses, we extended the Fujaba Tool Suite RE with two plug-ins. The first plug-in, called MetricsCalculation and described in Section 2, offers the calculation of several object-oriented software product metrics. The second plug-in PolymetricViews, described in Section 3, allows for viewing the metric results calculated by the first plug-in in polymetric views as introduced by Lanza. The paper closes with some future work issues.

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

2. METRICS CALCULATION

Before the MetricsCalculation plug-in is able to calculate software product metrics for certain model elements, the system to be analyzed has to be parsed into the Abstract Syntax Graph (ASG) representation. Therefore the MetricsCalculation plug-in uses the JavaAST plug-in and the JavaParser plug-in. The ASG comprises UML elements such as classes, attributes and methods as well as elements corresponding to classical syntax trees such as literals or assignments. Whereas the UML elements are used to represent declaration parts, the other elements are used to represent method bodies. In the following we call the whole representation the model of the source code.

Each metric has a unique acronym, e.g. LOC which stands for Lines Of Code or NOC, Number Of Children, which is the number of direct sub classes of a class. The user may select the metrics to be calculated from the list of all supported metrics (cf. Table 1). Each metric value together with its acronym is stored in a separate result object that is linked to the corresponding model element via Meta-Model Integration (MMI) pattern [2]. The plug-in offers to present all results in a table.

2.1 Contributing a new metric

Far more metrics exist than currently are supported by the plug-in. Therefore, the plug-in was designed to be easily extended by new metrics. Each metric can be calculated for a particular type of model element. LOC, for example, is calculated for a method, the metric WLOC computes the lines of code for a whole class, and the number of children (NOC) is calculated on classes as well. For each metric, a calculator class exists which takes a model element of the appropriate type as input and calculates its metric value. All calculator classes implement a common interface. The data about available metrics is stored in an XML file. The file contains an XML element for each metric, which besides the unique acronym contains a name, a description, and the fully qualified name of the calculator class. Thus, in order to contribute a new metric, a calculator class that implements the common interface must be developed and an element describing the metric has to be added to the XML file.

2.2 Metric thresholds

For each metric, the user may additionally configure a threshold. If the metric value of a model element exceeds this threshold, an annotation is created which connects the model element and the result object (again via MMI pattern). Currently, the annotations are only visible in class

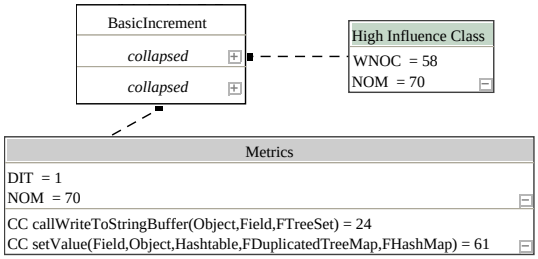

Figure 1: A class diagram with metric annotations.

diagrams. The shape of an annotation is a rectangle labeled with "Metrics". The rectangle is connected to the shape of the class containing the annotated model element (cf. Figure 1). In addition, if a class contains several model elements with metric annotations, only one annotation which is a union of all metric annotations is shown.

The results shown in Figure 1 were produced by first parsing the source code of the *basic*, *asg*, and *uml* packages of Fujaba (including sub packages). Afterwards, the metrics DIT, NOM and CC were selected to be calculated and annotated with thresholds 0, 20, and 20, respectively. The results show that the class BasicIncrement is at the first level in its inheritance hierarchy (DIT = 1) and defines 70 methods (NOM = 70). It has two methods with cyclomatic complexity (CC) 24 and 61, respectively. All other methods have CC values lower than 20. According to [9], CC values from 11 to 20 are still acceptable whereas higher values should be avoided.

2.3 Metric combinations

Single metric values are sometimes not very expressive. The fact, for example, that BasicIncrement has 70 methods, is not too edifying but not too interesting either. However, BasicIncrement has also a rather high number of all descendant classes (WNOC), i.e. 58. The 58 subclasses of BasicIncrement inherit its 70 methods which means that BasicIncrement has a high influence in its inheritance hierarchy. Note that the WNOC value was calculated when only the source files in the *basic*, *asg*, and *uml* packages of Fujaba (including sub packages) were parsed. The WNOC value of BasicIncrement would be much higher if Fujaba had been parsed completely.

To detect combinations of certain metric values, we offer the specification of boolean expressions over the values calculated by (possibly different) metrics for the same model element. A metric combination for high influence classes could be specified as e.g. (WNOC $\geq$ 10) & (NOM $\geq$ 20). In metric combination expressions the logical operators AND (&), OR (|), and NOT (!) may be used to join an arbitrary number of terms, possibly nested with parentheses. Each term may compare a metric value, represented by its acronym, with a number. As comparison operators ==, $<$, $\leq$, $>$, and $\geq$ are supported.

If a metric expression evaluates to true, an annotation is created and linked to the respective model element. The annotations created by metric combinations are displayed on class diagrams as well. They are also rendered as rectangles labeled with a name for the combination, e.g. High Influencing Class (cf. Figure 1), and linked to the class containing the annotated model element. In addition, the annotation shows the values of all metrics involved in the combination.

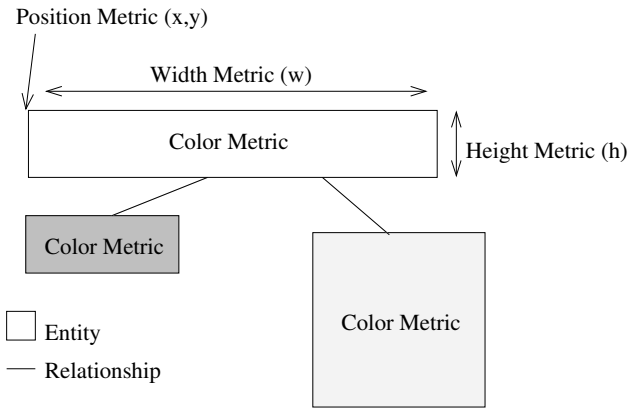

Figure 2: Up to 5 metrics can be visualized for one entity. In addition, entities may have relationships that do not carry metric values.

3. METRICS VISUALISATION

Metrics are numbers representing facts. We use class metrics such as number of methods (NOM) or number of attributes (NOA) or method metrics such as number of parameters (NOP). At the end of the previous section we argue that combinations of metric values are more expressive than a single value. We used a certain combination to give some hints to interesting parts in a software system, i.e. high influence classes. This approach has some drawbacks. We firstly have used absolute values and secondly the values are based on experience. Whereas normalization is a solution for the first problem, the second still remains, namely the definition of what are runaways in the context of the actual software system to be analyzed. In [6] Lanza presents Polymetric Views, which provide a visual representation of metric combinations. Detecting runaways is done manually by a developer looking at the produced pictures. Starting from the work presented in [6], we developed a Fujaba plug-in to visualize metric values as polymetric views.

3.1 Polymetic Views

Polymetric views are two-dimensional graphs containing nodes and sometimes edges, which are arranged in a certain layout. Nodes represent entities of the analyzed software system and edges represent relationships between the entities. Each node is a rectangle and can carry up to 5 metric values depending on the certain layout, whereas edges do not carry any metric information. Figure 2 illustrates the 5 potential metrics of an entity:

- **Size:** Either the width and height of a node (w,h) can represent a metric value. Both values have to be greater than 0.
- **Color:** The color of a node may also represent a metric. Valid values are one of 256 gray-scale values from black to white.
- **Position:** The position of a node (x,y) are the forth and fifth possible metric values of an entity. This assumes that the used layout allows for positioning nodes freely.
- **Layout:** Lanza proposes five major layout algorithms: Tree, Scatterplot, Histogram, Checker, and Stapled. Our prototype currently provides Tree and Checker layout.

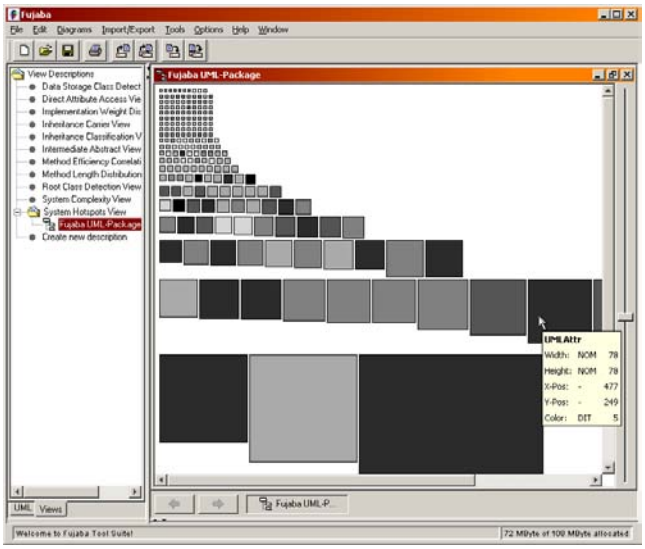

Figure 3: System-Hot-Spot view of the de.uni-paderborn.fujaba.uml package.

The first considers relationships as hierarchical order to arrange nodes and the second places nodes with the same metric value in the same row or column.

Absolute metric values may be problematic in polymetric views. High metric values used for the size or the position of a node may result in views where only a limited part is visible on the screen. For example a white colored node with a width and height larger than 3-5 times of the screen size may result in a white screen showing the inner part of the node only. Thus, a developer is not able to get an overview of all nodes in the view. To solve this problem, we use a mapping function that maps metric values to values better suited for the screen-size. Mainly the mapping function scales absolute metric values. To let the developer see all entities even the ones with metric values of 0, we add a constant value (minimal node size) to the scaled metric values. Both values, the zoom factor as well as the minimal node size value are interactively changeable for a certain view.

3.2 Prototype

Entities of a polymetric view correspond to parts of the software system currently under investigation. A certain polymetric view is an instance of a polymetric view description on a certain part of the software, called polymetric view context. Hence we currently focus on coarse-grained analysis to get a first impression of the software, the context is a set of classes. Our prototype consists of a dialog to select the context based on the current package structure of the software.

Figure 3 shows the prototype of the PolymetricViews plug-in. The project tree on the left hand side shows all currently available polymetric view descriptions. The current view is the System-Hot-Spot view of the core meta-model classes of Fujaba located in the *uml* and all sub packages.

Each of the 203 entities in the polymetric view corresponds to one class. The width and height of a node carry the number of methods of the class (NOM) and the color carries the class' depth in the inheritance tree (DIT). The checker layout arranges the nodes according to their size in ascending order. Due to the 2-column layout of this paper, each line has a fixed number of 10 nodes, except the last line.

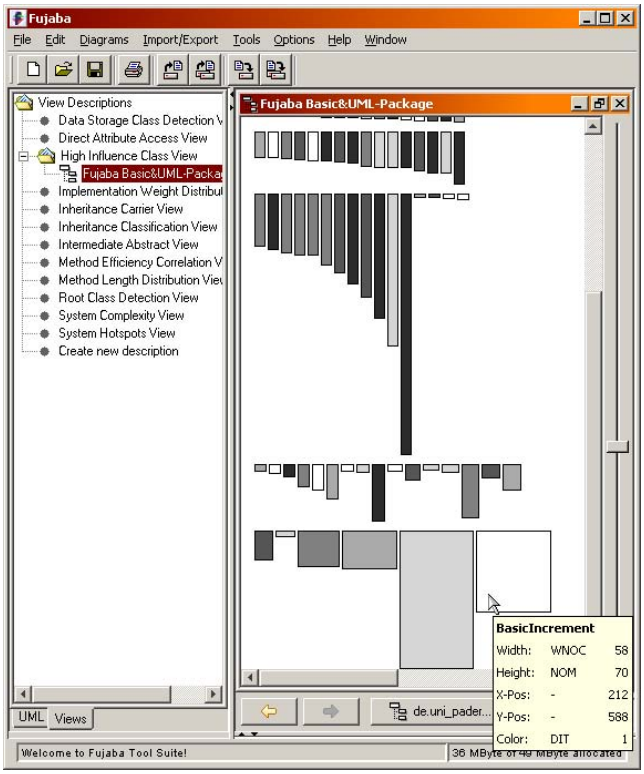

Figure 4: High-Influence-Class view example.

The zoom value can be modified with the slider on the right hand side. Going over a node, the appearing tool tip note shows the absolute metric values of the entity. For example, the UMLAttr class has 78 methods and 4 super classes in the inheritance tree.

What is the interpretation of this polymetric view? The meta-model classes of Fujaba have usually small interfaces, i.e. number of methods. The UMLAttr entity is at a depth of 5 in the inheritance tree. The black color of the UMLAttr entity indicates that this is also the maximum depth of the tree. Furthermore, there is no really dominating color, which would indicate an inheritance level with many classes. More problematic is the rightmost node in the last line. The node that corresponds to the UMLClass class is nearly double sized compared to the next smaller one, which means the entity has nearly twice the number of methods. Such classes in general need further analysis. In this case we observed that UMLClass is the central part of the meta-model of Fujaba and has many associations to other classes in the meta-model. Hence we originally generated the class and map associations to access methods, it has a huge number of association access methods. Thus it would be better to count only non-access methods. Unfortunately software product metrics are inappropriate to classify methods in that way. For this purpose, the already existing pattern instance detection is better suited.

3.3 Defining Polymetric Views

Lanza proposes 12 polymetric views organized in 3 categories for a coarse-grained analysis of a system. The first category is titled *First Contact Views* such as the *System Hot Spot* view shown in Figure 3. The second category *Inheritance Assessment Views* contains views to analyze the

inheritance structure. Views in the third category *Candidate Detection Views* detect entities that need further analysis.

In Section 2 we have proposed a metric combination of the number of methods (NOM) metric and the number of all descendant classes (WNOC) metric. Classes with metric values (WNOC $\geq$ 10) & (NOM $\geq$ 20) got a *High Influencing Class* annotation. Non of Lanza's polymetric views offers this metric combination, thus we have to define a new one.

The PolymetricViews plug-in allows the developer to dynamically define new view descriptions. A new polymetric view description consists of the following parts:

- the assignment of metrics to the size, position and color of an entity.
- a layout that arranges the entities and relationships.
- a factory that provides entities and relationships.

After the developer has defined the new polymetric view description, new views can be created. For example, Figure 4 shows the above described combination of the NOM and WNOC metrics of classes. The largest sized node is the BasicIncrement class. To our surprise the second largest sized node corresponds to class UMLIncrement that is located 2 inheritance levels below the BasicIncrement class. The UMLIncrement entity has a WNOC value of 56 that is 2 less than the BasicIncrement entity but much more methods, i.e. 195. We can not make the statement that this is ugly design, because the methods in UMLIncrement may override the ones in BasicIncrement. To strengthen an ugly design statement we have to make further investigation, perhaps enhance the new polymetric view description with the number of overridden methods (NMO) metric.

4. FUTURE WORK

Primary future work is the seamless integration of the metric analysis techniques into the overall reverse engineering process supported by the Fujaba Tool Suite RE. In particular, the existing pattern instance recognition and the metrics calculation will be integrated in such a way, that certain metric values or combinations may be used as triggers for fine-grained analyses with the pattern recognition. Furthermore, pattern specifications will be enabled to require that the metric values calculated for certain model elements do (not) exceed specific threshold values. The polymetric views will be used to determine those threshold values for the actual system to be analyzed.

Acknowledgments

We thank Lukas Roth and Jens Falk who implemented the MetricsCalculation and the PolymetricViews plug-ins, respectively, as part of their bachelor theses.

Acronym	Short description	Scope
ADIT	Attribute Depth of Inheritance Tree	Attribute
AvgCC	Average cyclomatic complexity	Class
AvgNLA	Average number of local accesses	Class
CC	McCabes cyclomatic complexity	Method
DIT	Depth of inheritance tree	Class
LCOM	Lack of cohesion in methods	Class
LOC	Lines of code in method	Method
MDIT	Method depth of inheritance tree	Method
NAM	Number of abstract methods	Class
NBLD	Nested block depth	Method
NCV	Number of class variables	Class
NI	Number of invocations	Method
NIA	Number of inherited attributes	Class
NIV	Number of instance variables	Class
NLA	Number of local accesses	Attribute
NMA	Number of methods added	Class
NMAA	Number of accesses on attributes	Method
NME	Number of methods extended	Class
NMI	Number of methods inherited	Class
NMO	Number of methods overridden	Class
NOA	Number of attributes	Class
NOC	Number of children	Class
NOCL	Number of classes	Project
NOINT	Number of interfaces	Project
NOM	Number of methods	Class
NOP	Number of parameters	Method
NOS	Number of statements	Method
NPA	Number of public attributes	Class
PLOC	Lines of code in project	Project
SIX	Specialization index	Class
WLOC	Lines of code in class	Class
WMC	Weighted methods per class	Class
WNI	Number of all method invocations	Class
WNLA	Sum over NLA	Class
WNMAA	Sum over NMAA	Class
WNOC	Number of all descendant classes	Class
WNOS	Number of statements in class	Class

Table 1: Currently supported metrics. Each metric has a unique acronym, a name, and a scope, which indicates the type of ASG element the metric can be calculated for.

5. REFERENCES

[1] W. Brown, R. Malveau, H. McCormick, and T. Mombray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, Inc., 1998.

[2] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. 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)*, 6(3):203–218, Aug. 2004.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[4] N. E. Fenton and S. L. Pfleeger. *Software Metrics - A Rigorous & Practical Approach*. International Thompson Computer Press, second edition edition, 1996.

[5] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[6] M. Lanza. *Object-Oriented Reverse Engieering*. PhD thesis, University of Berne, Switzerland, 2003.

[7] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[8] J. Niere, W. Schäfer, J. P. 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.

[9] Software Engineering Institute, Carnegie Mellon University, USA. *Cyclomatic Complexity: Software Engineering Roadmap. Online at http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html*.

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

A Plugin for Checking Inductive Invariants when Modeling with Class Diagrams and Story Patterns*

Basil Becker, Holger Giese, and Daniela Schilling†
Software Engineering Group, University of Paderborn, Germany
[basilb|hg|das]@uni-paderborn.de

ABSTRACT

Systems with complex structures that change at run-time can be specified with UML class diagrams and Story Patterns. Available verification techniques, which can check required safety properties automatically, only cover less expressive behavioral descriptions or finite state models of moderate size. In [4], an approach to check whether a given safety property is an inductive invariant of the system has been developed where no such restrictions apply. Within this paper we describe our Fujaba Plugin which realizes this approach by means of an example and present some first evaluation results for the Plugin.

1. INTRODUCTION

UML class diagrams and Story Patterns [3], as supported by the core of Fujaba, can be employed to model and simulate systems where the structure of the system is subject to frequent changes. However, if safety properties have to be ensured, the rather strong expressiveness of the underlying typed graph transformation systems renders an automatic verification impossible in the general case.

Formal verification techniques to check required safety properties automatically for systems with possibly infinite state spaces are only available for less expressive modeling techniques and usually provide only approximations (cf. [1]). The problem can be addressed with model checking if the employed models have a finite state space of moderate size (cf. [7, 6]).

Therefore, we have developed an approach to check whether a given safety property is an inductive invariant of the system in [4] which supports Story Patterns and is applicable without considering the possibly infinite set of relevant system configurations. Within this paper we describe the Fujaba Plugin which supports this approach.

The paper initially outlines the modeling concepts employed in the approach in Section 2. In Section 3 the background for the verification task is described and the verification of the example with the Plugin including the generation of a counterexample is outlined. The architecture of the Plugin and first evaluation results for the Plugin are presented in Section 4. The paper is closed with some final conclusions and an outlook on future work.

†Supported by the International Graduate School of Dynamic Intelligent Systems at the 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.

2. MODELING

This Section sketches our modeling concepts. They are explained by using a simplified example taken from the Special Research Initiative 614 “Self-optimizing Concepts and Structures in Mechanical Engineering”. The example are autonomously driving shuttles which can save energy by building contact free convoys while driving. For these shuttles software has to be developed that meets several safety properties.

In a first step, the physical domain is considered and all elements for which software has to be developed or which influence those elements are modeled as classes in a class diagram which is called the system's ontology. The dependencies between these elements are modeled as associations. Figure 1 depicts such a diagram. Software has to be developed for the Shuttles and the BaseStations which monitor the railway system. The railway system is modeled by Tracks which are connected by a successor link.

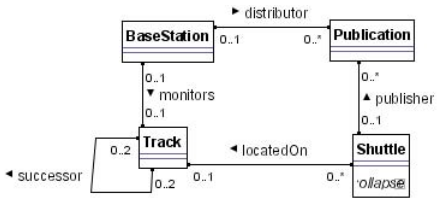

Figure 1: Ontology of the system

As the system is safety-critical the communication between several elements has to follow strict protocols which are also specified as classes in the diagram. The class Publication is such a protocol. It provides the communication between a Shuttle and a BaseStation. Each shuttle driving on a certain track has to be registered at that BaseStation which monitors the track the shuttle is located on. A shuttle is registered at a BaseStation if the Publication protocol is instantiated between them. The protocol ensures that the BaseStation gets to know if a shuttle failed and in this case warns the other shuttles.

After all relevant elements are identified and added to the ontology the system behavior can be specified. The behavior is described by Story Patterns.

A Story Pattern describes how a system state given as instance graph is modified. A Story Pattern consists of several objects and links which can be annotated with $\ll destroy \gg$ or $\ll create \gg$. To apply a Story Pattern all elements (objects and links) which are not annotated or annotated with

$\ll$ *destroy* $\gg$ have to be mapped to the instance graph. If this is possible all elements annotated with $\ll$ *destroy* $\gg$ are removed from the instance graph. Then all elements which are annotated with $\ll$ *create* $\gg$ are added to the instance graph. An example for a Story Pattern is given in Figure 2. This Story Pattern, called move, models the movement of a shuttle from one track to the following one. Thereto the locatedOn link between the shuttle s and the track ta is deleted and a new locatedOn link between the shuttle and the track tb is created.

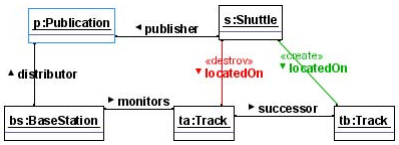

Figure 2: Story Pattern move describing a shuttle's movement

Figure 3 gives another example for a Story Pattern. It describes the instantiation of a Publication protocol. The crossed out elements require that the instantiation may only take place if the protocol is not yet instantiated between a certain pair of Shuttle and BaseStation. The crossed out elements are called negative elements. The Story Pattern may only be applied to an instance graph if all positive elements can be mapped to elements of the instance graph but the negative ones cannot.

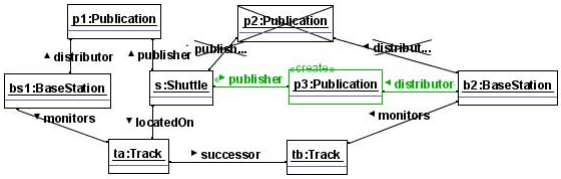

Figure 3: Story Pattern instPub describing the instantiation of the Publication protocol

The behavior of systems, like e.g. the shuttle system, has to be developed in a way that hazards are avoided. A hazard is a critical situation, like e.g. the impending of a shuttle collision. Such a hazard can itself be described by a Story Pattern. In our example one possible hazard is that a shuttle is not registered at the BaseStation which monitors the track it is located on. In this case the BaseStation would not be able to detect if the shuttle failed and therefore could not warn the other shuttles. The Story Pattern describing this hazard is given in Figure 4.

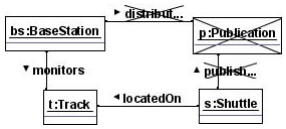

Figure 4: Story Pattern describing a hazard

A hazard occurred if the Story Pattern describing the hazard can be applied to the instance graph.

As a Story Pattern describing a hazard must never be applicable they are called forbidden patterns.[1]

[1]In the following we use the term forbidden pattern if we are concerning Story Patterns that describe hazards and by Story Pattern mean those Story Patterns that describe system behavior.

3. VERIFICATION

In this Section we show how to verify that the Story Patterns do never create instance graphs that contain any forbidden pattern. A system state is correct if it does not contain any forbidden pattern. The system itself is correct if only correct states are reachable.

In general such systems have so many states that it is not possible to check for every reachable state that it is correct. A system state is reachable if there is a sequence of Story Patterns that are applied to the initial instance graph and result in the considered state. In other words such techniques check whether a state can be reached that is incorrect w.r.t. given forbidden patterns.

One specific case of system properties to be checked are invariants. An invariant makes a statement on all (reachable) states. There are two different kinds of invariants, operational and inductive ones. An operational invariant is a property that is fulfilled by the initial state and each state that is reachable from this initial state also fulfills the property. Considering the above given forbidden patterns the absence of these forbidden patterns is an operational invariant if the initial state does not contain any forbidden pattern and each reachable state does neither.

A stronger form of invariants are the inductive invariants. A property is an inductive invariant of a system if each correct state can only be transformed into other correct states, regardless whether the first state is reachable at all. If in addition the initial state is correct then an inductive invariant is also an operational one.

To check whether a system's behavior is correct one has to check whether each of the reachable states are correct, i.e. the absence of any forbidden pattern is an operational invariant of the system. Such an analysis suffers from two problems. At first the set of initial instance graphs can contain arbitrarily many graphs. For each of these initial graphs the reachability analysis has to be performed which is usual not feasible. Second, systems specified by Story Patterns can have infinite many reachable states and thus such a reachability analysis is impossible. Instead we check whether the absence is an inductive invariant. Thereby we do not have to consider the reachability of certain states.

There are two cases how a Story Pattern can transform a correct instance graph into an incorrect one. At first, there is no forbidden pattern which can be mapped to the instance graph because some of the forbidden pattern's positive elements are missing but the application of the Story Pattern creates these missing elements. Second, there is a forbidden pattern with negative elements for which the positive elements can be mapped to the instance graph but in addition also one of the negative elements can be mapped too. In this case the forbidden pattern did not occur so far, but the application of the Story Pattern deletes all elements from the instance graph to which the negative elements of the forbidden pattern are mapped.

As there can be also infinite many of such transformations we are interested in representatives that show the invalidation of a property. To find such representatives we exploit the fact that the application of a Story Pattern has only local effects. Therefore it is not necessary to consider complete graphs rather than parts of them. To do so we build so called target graph patterns (tgp) each of which represents a possibly infinite set of graphs. Such a tgp is built by merg-

ing a forbidden pattern with a Story Pattern[2]. The result is a pattern representing all those incorrect instance graphs which contain the pattern as a subgraph. To this pattern the Story Pattern is applied in back-wards direction which again results in a pattern, called the source graph pattern (sgp). If this sgp does not contain any of the forbidden patterns a witness is found that the considered Story Pattern can be responsible for the creation of a forbidden pattern. Otherwise, the application of the Story Pattern transforms an incorrect pattern into another incorrect one which is of no interest.

Consider the Story Pattern move and the forbidden pattern given above. One possible tgp for these two Story Patterns is given in Figure 5. To this tgp the Story Pattern is applicable in back-wards direction by construction. The sgp resulting from the Story Pattern application is depicted in Figure 6. This sgp is correct as the forbidden pattern cannot be applied to it. Thus a witness is found which indicates that the application of the Story Pattern move can produce an instance graph containing a forbidden pattern. A forbidden pattern will occur in a system if there is a reachable instance graph that contains the sgp and the move Story Pattern is applied to this instance graph. Then the resulting graph will contain the tgp and contains therefore the forbidden pattern. Thus our verification technique delivers a counterexample showing what can go wrong. This counterexample consists of two graph patterns and one Story Pattern.

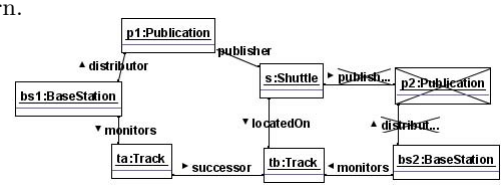

Figure 5: Example for a target graph pattern

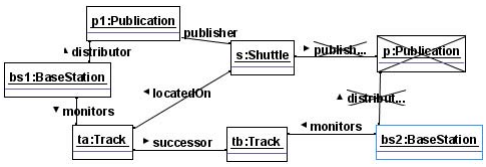

Figure 6: Example for a source graph pattern

After the mistake has been identified the corresponding Story Pattern can be adjusted such that the forbidden pattern cannot occur anymore. In our example we have to ensure that the Publication protocol is running before the shuttle is moving to a track which is monitored by a new BaseStation. This adjustment is performed by extending the move Story Pattern by additional objects of type BaseStation and Publication. Now the Story Pattern may only be applied if the Publication protocol is instantiated with the BaseStation which monitors the succeeding Track. The corrected Story Pattern is depicted in Figure 7.

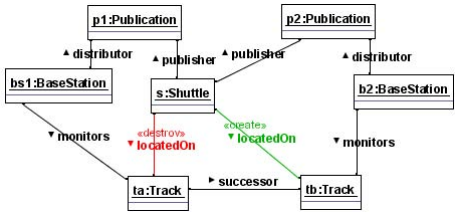

Figure 7: Story Pattern describing the corrected shuttle movement

4. THE PLUGIN

The plugin presented in [2] only depends on the Fujaba Kernel (version 4.2.1 or later).

To verify that a system given as set of Story Patterns is correct w.r.t. a set of forbidden patterns there are two main sub algorithms, one to merge Story Patterns and one to find matching of Story Patterns.

Merge-algorithm. The Merge-Algorithm is used to build the tgps. To build the pattern a partial matching m, which maps some elements (objects or links) of a Story Pattern to a forbidden pattern, is needed as input. The algorithm builds the tgp by extending the forbidden pattern by all elements of the Story Pattern which are not mapped by m to any element of the forbidden pattern.

Matching of Story Patterns. The Matching-Algorithm is needed to find all matching between a Story Pattern and a forbidden pattern.

The algorithm starts with two objects, one of the Story Pattern and one of the forbidden pattern. For these objects it is checked whether they can be matched, i.e. they are of the same type, either both positive or both negative and are not matched so far. If this is a valid matching it has to be tested whether all links which are incident to both of these objects can be matched too. Two links can be matched if the matching fulfills the same requirements as in the case of objects and additionally if the links are adjusted in the same direction. In the case that this test delivers a positive result the algorithm succeeds recursively with the pair of objects which is incident to the links but not visited before. If the test delivers a negative result some backtracking steps have to be performed.

Verification Algorithm. The Verification-Algorithm decomposes all Story Patterns into all their possible sub Story Patterns. Doing so negative objects are treated as non negatives. For each sub Story Pattern its matching with all forbidden patterns is calculated using the algorithm given above. If such a matching can be found it serves as input for the Merge-Algorithm which then computes the tgp. To this tgp the corresponding Story Pattern is applied in back-wards direction which results in the sgp. The algorithm checks for each forbidden pattern whether there is a matching which maps the forbidden pattern to the sgp. If no such matching can be found the algorithm tries to find

[2]In the case that a property is invalidated by creating some elements the building of the tgps is quiet similar to the gluing presented in [5]. While we check that a given rule set is correct, in [5] a correct rule set is derived by adding potentially exponentially many morphisms to its application conditions. Such additional conditional constraints of the application condition might contradict given system constraints and have to be checked at runtime. They can even result for rule sets that are checked as correct by our approach, due to our additional check of the resulting sgp for forbidden patterns.

another Story Pattern which is applicable to the sgp and which has a higher priority then the currently considered one. If there is no such other Story Pattern which prevents the currently considered Story Pattern a witness is found that the application of the considered Story Pattern can produce an incorrect instance graph. The verification stops and the algorithm outputs the found counterexample. In the case that either for each sgp a matching can be found that maps a forbidden graph to it or if otherwise a Story Pattern with a higher priority can be found which is proved to be correct the verification delivers that the considered system is correct w.r.t. the set of forbidden graphs.

For the above given algorithms we roughly estimated their complexity and evaluated them on several test cases. For the evaluation we used an AMD Athlon Processor with 1.4GHz and 512MB working capacity.

The exact calculation of the matching algorithm's complexity is hard to determine, as it is a highly recursive algorithm. In [2] a rough estimate has been done. Assume that G is the Story Pattern and H is the forbidden pattern. $\Delta(G)$ is the highest degree of all objects n in G and d is the depth of the search tree the algorithm traverses. Then the complexity is about: $O(|V_G|*|V_H|*\Delta(H)^{\Delta(G)*d})$. This estimation has been calculated for the worst case where all objects and links have the same type.

The complexity presented above is a theoretical result that does not allow to give evidence about the time the algorithm needs to complete its computation. Therefore the matching algorithm has been evaluated in [2] using untyped graphs.

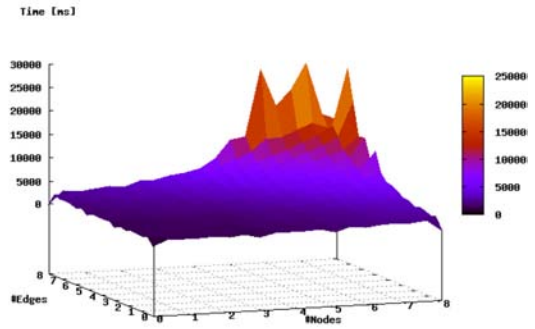

Figure 8: Diagram that shows the evaluation of the graph matching algorithm

We took one Story Pattern, representing a complete graph of order 3, and computed all matchings between this Story Pattern and several randomly chosen forbidden patterns of different size. At each run the number of objects and / or links was increased step by step. The larger the randomly created forbidden patterns became, the longer the computation lasted. Figure 8 displays the results we found out.

The Merging-Algorithm's complexity is linear in the size of the both Story Patterns that have to be merged. Let $G = (O_G, L_G)$ and $H = (O_H, L_H)$ be these two Story Patterns, with O is the set of objects and L the set of links. The complexity then can be expressed as follows: $O(|V_G|+|E_G|+|V_H|+|E_H|)$. This estimation is very obvious as it is clear that the algorithm has to visit each item of both graphs only once.

The verification algorithm's complexity is mainly determined by two parts. At first the number of Story Patterns and the number forbidden patterns, and second the size of the single patterns. The complexity is quadratic in the number of the Story Patterns and forbidden patterns and exponential in the maximum size of a single pattern. The exponential complexity is indebted by the fact, that the verification algorithm invokes the matching algorithm.

The complexity shown above is a worst case complexity. To prove the adaptability of our approach we modeled and verified a system in [2]. The result of this evaluation shows that it is possible to verify a system consisting of four Story Patterns and three forbidden patterns in an average time of 19 sec.

5. CONCLUSION AND FUTURE WORK

We have demonstrated that the outlined modeling approach in combination with the presented Plugin enables the systematic development of safe infinite state systems. The generation of a counterexample guides the designer in the stepwise correction and refinement of the structural and behavioral model. While the employed algorithms show rather high worst-case complexity, the presented premature evaluation results for the Plugin indicate that in practice the approach is often feasible.

In future work, we want to improve the checking engine employed in the Plugin and also extend the expressiveness of our approach also taking attributes, time, and hybrid behavior into account.

REFERENCES

[1] P. Baldan, A. Corradini, and B. König. Static Analysis of Distributed Systems with Mobility Specified by Graph Grammars - A Case Study. In H. Ehrig, B. Krämer, and A. Ertas, editors, *ISPT Conference Proceedings*, 2002.

[2] B. Becker. Automatische Überprüfung induktiver invarianten für graphtransformationssysteme. Bachelor's thesis, University of Paderborn, Paderborn, Germany, 2005. german.

[3] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, November 1998.

[4] H. Giese and D. Schilling. Towards the Automatic Verification of Inductive Invariants for Invinite State UML Models. Technical Report tr-ri-04-252, University of Paderborn, Paderborn, Germany, December 2004.

[5] R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars - a constructive approach. In *Proc. of SEGRA-GRA'95 Graph Rewriting and Computation"*, ENTCS, 1995.

[6] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS)*, 2003.

[7] D. Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, May 2004.

Formal Verification of Java Code Generation from UML Models

Jan Olaf Blech Sabine Glesner Johannes Leitner

Institute for Program Structures and Data Organization
University of Karlsruhe, 76128 Karlsruhe, Germany

ABSTRACT

UML specifications offer the advantage to describe software systems while the actual task of implementing code for them is passed to code generators that automatically produce e.g. Java code. For safety reasons, it is necessary that the generated code is semantically equivalent to the original UML specification. In this paper, we present our approach to formally verify within the Isabelle/HOL theorem prover that a certain algorithm for Java code generation from UML specifications is semantically correct. This proof is part of more extensive ongoing work aiming to verify UML transformations and code generation within the Fujaba tool suite.

Keywords

Statecharts, Fujaba, Isabelle/HOL, verification, semantics, transformations, code generation.

1. INTRODUCTION

The generation of code from specification languages like UML is an important aspect of the Model Driven Architecture (MDA). State-of-the-art Computer Aided Software Engineering (CASE) tools like the Fujaba tool suite come with such code generation mechanisms. To ensure correct software and system behavior, it is a necessary prerequisite that the semantics of the original UML systems is preserved during code generation.

To ensure that such transformations are correct, formal verification is necessary. In this paper, we consider an algorithm that transforms simplified Statecharts into a Java-like language. We verify that this transformation algorithm is correct. Therefore we formalize the semantics of Statecharts and the targeted Java subset as well as the transformation algorithm within Isabelle/HOL. Furthermore, we prove, also within Isabelle/HOL, this code generation algorithm correct by showing that each Statechart specification and the corresponding program code are bisimilar. The use of bisimulation as equivalence criterion between Statecharts and the Java-like programming language ensures an adequate semantics formalism even for non-terminating systems and programs. In order to obtain machine-checked proofs as well as reusability of proofs, our proofs and formalizations are conducted within Isabelle/HOL.

This paper describes ongoing work that is part of a larger project aiming to verify real life transformations from UML specifications of CASE tools to Java code. Apart from our already finished formalization and proof work, we give a detailed plan for our future work concerning verification of MDA transformations. We believe that verification of Fujaba transformations can be a major and highly interesting research area in the future.

In Section 2, we discuss various approaches for the formalization of Statecharts. After that, we explain basic foundations for behavioural equivalence proofs in Isabelle/HOL in Section 3. Our formalization of the Java-like programming language as well as our Statechart formalization together with the actual correctness proof is discussed in Section 4. We discuss related work in Section 5. In Section 6, we conclude and present our future workplan.

2. SEMANTICS OF STATECHARTS

Statecharts are an extension of finite state machines which are implemented in many tools and widely used in practice. Nevertheless, the definintion of their semantics poses subtle difficulties due to inherent ambiguities. In this section, we first introduce Statecharts in Subsection 2.1. Afterwards, in Subsection 2.2, we summarize how executable code, e.g. Java code, can be generated from them automatically. In Subsection 2.3, we explain the difficulties when formally defining the semantics of Statecharts. Finally, in Subsection 2.4, we discuss methods to prove equivalence of Statecharts.

2.1 The Statechart Language

Statecharts [12] are a visual language which enhance finite state machines by hierarchical and parallel composition of states (and broadcast communication between parallel transtitions). They have found wide-spread use in the modelling of complex dynamic behaviour and are part of the UML standard [20] with advanced features such as history mechanisms and extended transition trigger conditions. In UML Statecharts, transitions as well as states can be decorated with actions, i.e. statements in some imperative language, thereby significantly increasing the expressive power of statecharts.

2.2 Code Generation From Statecharts

There are numerous tools providing code generation from Statecharts. Most of these employ one of the following three strategies of code generation which are almost directly derived from code generation strategies for finite automata.

- *(Hierarchical) Switch/Case Loop* This most simple approach creates a nested switch/case statement that

branches according to the current state and the current event. Within a branch, transition-specific code, i.e. the action associated with the transition, is executed and the current state is set to the target state of the transition. Hierarchical and concurrent structures can be achieved using recursion.

- *Table-driven approach* The second approach stems from a well-known method to implement finite state machines in compiler construction (e.g. scanner generation by the well-known unix tool "lex"). The actions caused by an event in a specific state are stored in a (nested) state/input table. In its most basic form, entries in this table might only consist of output symbols and successor states. When more complicated actions are used, more complex structures are necessary for the representation of state table entries, as demonstrated in [27]. In principle, a table-based approach is also suited for hardware implementation in embedded systems.

- *Virtual Methods* Deeply nested switch/case blocks may not be desirable in an object oriented system. This is especially true when code generated from a Statechart is subject to manual modification and maintenance ("round-trip engneering"). An alternative method of code generation from Statecharts makes use of an extension of the state pattern [7]. In this method, each state becomes a class in an inheritance hierarchy created in parallel to the substate hierarchy of the statechart. The events consumed by these states are realized as virtual method calls to the respective state classes.

These are the basic strategies for code generation from Statecharts. A more detailed overview can be found in [27]. [26] shows how hierarchical structuring information can be exploited to obtain smaller and more efficient code following the table-based strategy.

In this paper, we present our verification of code generation that follows the first strategy. In future work, we aim to extend our correctness proof to also allow for the verification of other generation algorithms as well.

2.3 Formal Semantics of Statecharts

The behaviour of a Statechart is modeled by a transition system whose states correspond to the configurations of the represented Statechart. A configuration itself is a maximum set of Statechart states that can be active at the same time. The actual behavior of the Statechart lies in the state transition function, describing how one configuration leads to the next, depending on the current input symbol. A formal definition of this step transition function has proven somewhat challenging. A multitude of approaches has been taken to define such a semantics. A comparison between 17 of these approaches can be found in [25].

One reason for the difficulties in defining a state transition function is the desired property of *synchrony*. Synchrony means that the system should react immediately to incoming events. In particular, incoming events and resulting actions should happen without delay at the same time. Most

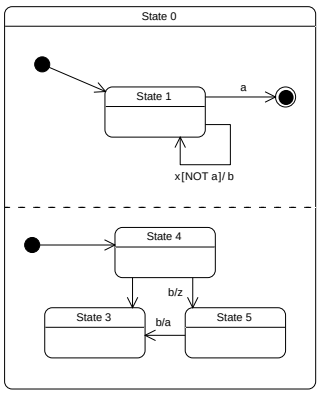

Figure 1: Event Conflict

of the time, this does not pose a problem physically, since the sampling rate of the real system is typically limited and significantly longer than the reaction time of the system. However, using synchrony, concurrent states and broadcast communications allow for a single event to trigger a chain reaction of multiple transitions, called *microsteps*, in "zero time", creating a situation that is inconsistent with the notion of *causality* (i.e. that original actions and the actions triggered by them cannot be distinguished).

As an example, consider the event `x` passed to the Statechart depicted in Figure 1 in the initial configuration (State 1, State 4). The system will take the loop at State 1, and also go from State 4 to State 5, since the first transition emits the event `b` and synchrony causes the system to react immediately. In the next step, if the input symbol is again `x`, this leads to a contradiction, since the transition from State 5 to State 3 causes the event `a`, in which case the original transition (the loop at State 1) should not have been taken.

The conflicts involving synchrony and negated conditions can be resolved in different ways. One way is to consider only *globally consistent* steps, which is intended in [22]. Global consistency is difficult to achieve, since their might be multiple sets of transitions for a signal even if the state machine is deterministic. Another, more direct solution is to take the *causal order* of microsteps into account and require that conditions guarding a transition only apply to events that occurred *before* the transition was taken. This kind of behavior is sometimes called *local consistency*. Local consistency leads to the paradox situation that although all microsteps take place at the same time, the order in which they occur is not irrelevant.

Since synchrony, although a desirable property for the modeling of real-time systems, causes these problems and often leads to counter-intuitive behaviour (especially when more complicated actions are allowed), more "practical" semantics, including the STATEMATE [13] and the UML [3] semantics, disregard this property and realize a step-by-step

behaviour, in which events and actions generated in one step do not become visible until the next step. In UML, this is sometimes called *run-to-completion-semantics* [3].

Defining a step function becomes even more problematic when we aim to develop a *compositional* semantics. In general, compositional semantics means that the semantics of a larger program or system can be derived from the semantics of its parts. Especially in the case of parallel composition, Statecharts are "noncompositional" in nature – the behaviour of the concurrent state can differ significantly from that of its substates when parallel transitions have overlapping actions and conditions. When being concerned with the equivalence of Statecharts, it is important to define an equivalence as a congruence with respect to the Statechart constructors. This specifically means that states that are regarded as equivalent should again yield an equivalent Statechart when composed with the same state and Statechart constructor. [24] and [17] demonstrate that such an equivalence relation cannot be defined by regarding only complete steps. Instead, the causal ordering of microsteps needs to become part of the semantics. [24] achieves this by constructing more complex labels in the resulting transition system, which then contain information on how the respective step was constructed out of microsteps. [17] use two different kinds of transitions, microstep transitions and σ-transitions, which determine the beginning and completion of a step, to incorporate information about causal ordering directly into the transition system.

2.4 Equivalences of Statecharts

The ability to prove the equivalence of Statecharts allows us to prove the correctness of elementary Statechart-to-Statechart-transformations. [6] present a collection of 23 such transformations, e.g. splitting a state or shifting a transition up and down the substate hierarchy. To prove the semantical correctness of such a transformation, a suitable Statechart semantics is needed. Since we are interested in local Statechart transformations, compositionality becomes an even more important issue here. In [18], a number of equivalences (which are originally introduced in [5]) is applied to the labelled transition systems defined by a slight variation of the semantics in [24]. Moreover, congruence properties are studied with respect to Statechart constructor operations. They show that (of the six investigated relations), the weakest relation between Statecharts that is still a congruence is bisimilarity between their corresponding labelled transition systems.

In this paper, we consider a restricted subset of Statecharts consisting of non-hierarchical automata without concurrency and show how their equivalence can be defined by bisimulation in Isabelle/HOL. In the following section, we introduce some basics about the theorem prover Isabelle/ HOL as well as our formulation of bisimulation. Afterwards, in Section 4, we use this formalization to prove the equivalence between the considered restricted set of Statecharts and the Java code generated from them.

3. PROOF FOUNDATIONS: BISIMULATION AND ISABELLE/HOL

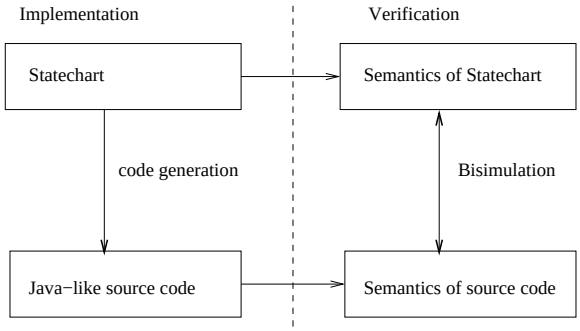

Figure 2: Verification of Code Generation

This section describes bisimulation as the method of choice for transformation verification. Moreover, we desribe some Isabelle/HOL related aspects.

3.1 Bisimulation

Our principle idea is to regard a Statechart and its transformation –a program in a higher programming language– as semantically equivalent if they denote the same observable behaviour. For example in a deterministic specification, they must have the same sequences of observable states. Figure 2 shows the principle to prove such a transformation correct. One needs a Statechart semantics and a target language semantics as well as mappings from concrete Statecharts and programs to their respective semantics. To verify a transformation, one has to show that the semantics of the original system is preserved during its transformation. In our case, this means that the sequences of observable states are the same for both Statechart and generated program. On the semantics side, this means that they have to bisimulate each other, i.e. that their semantics are in a bisimulation relation.

As a basic prerequisite, the semantics of Statecharts and Java-like programs must be comparable. For this purpose, we express their semantics as a Kripke structure.

DEFINITION 1 (KRIPKE STRUCTURES). *A Kripke structure is a five tuple (AP, S, R, S_0, L) where AP is a set of atomic propositions, S is a set of states, R is a transition relation, S_0 is the initial state, and L is a labeling function mapping states to sets of atomic propositions.* ◇

Hence, a Kripke structure is equivalent to an annotated state transition system.

DEFINITION 2 (BISIMULATION RELATION [4]). *Let $M = (AP, S, R, S_0, L)$ and $M' = (AP, S', R', S_0', L')$ be two Kripke structures with the same set of atomic propositions AP. A relation $B \subseteq S \times S'$ is a bisimulation relation between M and M' if and only if for all s and s', if $B(s, s')$ then the following conditions hold:*

1. $L(s) = L'(s')$

2. *For every state* s_1 *such that* $R(s, s_1)$ *there is* s'_1 *such that* $R'(s', s'_1)$ *and* $B(s_1, s'_1)$

3. *For every state* s'_1 *such that* $R(s', s'_1)$ *there is* s_1 *such that* $R'(s, s_1)$ *and* $B(s_1, s'_1)$ $\diamond$

We get a Kripke structure representing the semantics of a Statechart by unrolling its configuration transitions. These correspond to the transition relation R in the Kripke structure. The states S of a Kripke structure correspond to configurations in Statecharts with S_0 being the initial configuration. We encode a stream of upcoming input events as well as the current execution time in the Kripke structure states, too. A Kripke structure may have infinitely many states. This corresponds to a non-terminating Statechart.

We can describe the semantics of a program in a higher programming language as a Kripke structure M: If we specify its semantics such that the execution of a single instruction is atomic, the semantics of a program is specified by a state and a state transition function. Each state may consist of the current execution state and memory and variable mappings. Kripke structures are used as follows for the specification of program semantics: The atomic propositions represent the variable mapping, memory etc. S is the set of states reachable within the execution of the program M. R represents possible state transitions and the conditions under which they appear. S_0 is the initial state. L is a labeling function mapping states to their observable parts. This is appropriate for both Statecharts and programming language. Two programs, or a Statechart and a Java-like program, resp., are bisimilar if there exists a bisimulation relation B such that the initial states of both programs are within the relation.

If we describe the semantics of a Statechart as a Kripke structure M and the semantics of a corresponding program in a higher programming language as a Kripke structure M', then the bisimulation relation B expresses behavioral equivalence, with an equivalence criterion that we can choose freely: E.g. we can restrict the variables that appear in states – as well as in the atomic propositions – to input/output values. Then $L(s) = L'(s')$ checks state equivalence. With the notion of bisimulation, we have a formal criterion under which a program and a Statechart show the same behavior.

In the case of deterministic systems that we examine in this paper, the requirements for a bisimulation get even simpler: We regard two programs as semantically equivalent iff:

- They start with equivalent initial states s and s'. This is denoted $s \simeq_O s'$ where O is some set of observable actions. By equivalence we mean that the observable parts of the states must be the same, corresponding to the requirement $L(s) = L'(s')$ in Definition 2.

- For two states s and s' in the bisimulation relation, we require that the succeeding states are equivalent again. This is formalized in Isabelle/HOL as:
  $\forall\ s\ s'\ .s \simeq_O s' \longrightarrow next\ s \simeq_O next\ s'$
  where *next* returns the succeeding state.

This notion of program and Statechart equivalence captures very elegantly the semantics of both terminating and non-terminating programs and Statecharts. With its state abstraction, it is flexible enough to prove most transformations and optimizations correct. If we want to prove the correctness of a code generation algorithm from Statecharts, we have to show that Statechart and generated program(code) denote the same sequence of observable states which is exactly what a bisimulation proof does.

Bismulation may be defined on Kripke structures. A mathematically more elegant approach is to use coalgebraic data types instead of Kripke structures [14].

3.2 Isabelle/HOL-Specific Aspects

Isabelle/HOL is a generic higher order logic (HOL) theorem prover ensuring a very high expressivness of specifications. Theorem provers can be used to create specifications, formulate lemmata and theorems on them and prove them correct. Unlike model checking, working with theorem provers, especially those using higher order logics, is highly interactive. Specifications have to be designed very carefully in order to be able to prove them correct. The process of proving a system specification correct takes some effort but often reveals errors in the specification that would have been overlooked otherwise.

In Isabelle/HOL, bisimulation can be formulated in multiple ways. A very elegant way is to use coalgebraic datatypes, e.g. lazy lists [21], which in most cases of practical relevance come automatically with a bisimulation principle. In contrast to model checking, our Isabelle proofs verify complete semantical equivalence and not just certain aspects or conditions.

4. VERIFYING THE TRANSFORMATION FROM UML TO JAVA

In this section, we present our verification of Java code generation from UML models. Therefore, we consider a simplified subset of UML Statecharts, namely finite state machines (FSMs). To verify code generation, i.e. the transformation from an FSM to a target language program, a semantics for both FSMs and the target language with the same semantic domain is required. As already explained in the previous section, we concentrate on observational equalivalence by modelling semantics as the state transition sequences that can be observed during execution. In Subsection 4.1, we show how we represent FSMs and their semantics in Isabelle/HOL. Then, in Subsection 4.2, we introduce our target programming language WSC that contains while, switch, and case statements. The code generation algorithm is given in Subsection 4.3. Its correctness proof is presented in Subsection 4.4.

4.1 Formalization of Finite State Machines

Finite state machines are formalized as tuples (S, E). S is a list of *states* having arbitrary type. This allows in particular for hierarchical FSMs since the type of a state can be again an FSM. (Note that in the work presented here, we have not dealt with hierarchical FSMs. But since we want to so soon, we have already adjusted the specifications for this purpose.) E is a list of transitions connecting the states in S. A transi-

tion consists of its *source* and *target* (which are represented by the position of the source and target states in the state list S), a *trigger* symbol and an *action* symbol. We enhance the set of action symbols by the new symbol `silence`. With this extension, we make sure that every transition has an action associated with it.

Our decision to use a list to store the states of a Statechart has many benefits. It eliminates the need to find an ordering of states when generating code. Moreover, it is suitable for hierarchical automata since lists are inductive types. Active states can now be referenced by their position in the state list. We call the pairing of state machine and currently active state an *FSM configuration*. For example, (((State 4, State 3, State 5), ((1,2), (1,3), (3,2))), 1) is the initial configuration of the lower substate of the statechart in Figure 1.

The semantics of an FSM is the potentially infinite sequence of output symbols it produces given a sequence of input symbols. If the state machine is deterministic, i.e. there is at most one transition for each trigger and source state, a partially defined *step function* exists that selects this transition (if it exists) for a given state and input symbol. The output sequence is then calculated by the corecursive application of this function. (For simplicity, one might think of this corecursively defined output sequence as the potentially infinite list of transitions that are performed during the run of the FSM.)

4.2 The WSC Language

As target language, we consider a simplified subset of Java called WSC (while-switch-case) that contains switch and case statements and a specialized while loop. The language has only two variables of type integer, without the ability to define new ones. WSC only incorporates the limited functionality needed for the code generated from state machines, and can be easily transformed into real Java. Our definition of its syntax within Isabelle/HOL is given in Figure 3.

```
datatype WSC_Variable =
  CURRENTSYMBOL | STATE
  /— This language has only two variables/

datatype WSC_Expression =
  CONST nat | VAR WSC_Variable

datatype WSC_Statement =
  WHILE_NEXT_SYMBOL WSC_Statement |
  SWITCHCASE WSC_Expression
    "( WSC_Expression × WSC_Statement ) list"
    WSC_Statement
        ( "SWITCH _ CASES { _ } DEFAULT _" ) |
  ASSIGN WSC_Variable WSC_Expression |
  CONS WSC_Statement WSC_Statement ("_;_") |
  OUTPUT ActionType |
  SKIP
```

Figure 3: Syntax definition of the WSC language

The semantics of WSC is defined by specifying for each program a potentially infinite sequence of ouput symbols. For each pair consisting of a WSC program and its current state, we define the observable *output*, the *successor state*, and the *continuation*, i.e. the remainder of the program to be executed. Thereby, a state is a function mapping variables to values. We can then corecursively apply these functions, thus yielding a potentially infinite sequence of successor states and output symbols. For a more detailed description of this procedure, see [16]. Since we are only interested in output equivalence, we can disregard the sequence of states. Obviously most WSC statements do not produce any output, thus the output sequence will contain a lot of empty events which are not generated by the corresponding statechart semantics. We therefore define a *cleaned sequence* which contains only the elements that are not empty.

4.3 The Code Generation Algorithm

From an initial statechart configuration (((${s_1}, \ldots, {s_n}$), (t_1, …, t_n)), n) we generate WSC code according to the algorithm depicted in Figure 4.

```
while_next_symbol {
 switch ( STATE ) {
  .
  . – for j = 1...n
  case ( j ) {
   switch ( CURRENTSSYMBOL ) {
    .
    . – for all transtions t_k with source(t_k) = j
    case( trigger(t_k) ){
     STATE := target(t_k);
     output action(t_k)
    }
   }
  }
 }
}
```

Figure 4: Overview of Code Generation Algorithm

The default cases of the two switch statements are not shown – they are both empty: For the inner switch statement, this means that an input symbol has occurred that is not the trigger of any transition in the current state. The outer switch statement is never reached if the original state machine is reasonably well-formed, since this would require the STATE variable to point to a non-existing state.

4.4 Correctness Proof

This transformation from FSM to WSC is considered semantically correct iff the semantics of source (i.e. FSM) and target (i.e. WSC program) are always the same. Thus, we have to compare the output sequences of a finite state machine and its generated WSC code. We use out_{WSC}, out_{FSM} resp. to denote these output sequences. Apart from the FSM or WSC program they need an input parameter I – a stream of external events. Using the bisimulation principle from section 3 to show that two sequences are equal, we have to find and define a *bisimulation relation* in which they are contained. In this case, this means a relation $\sim$ such that:

(1) $out_{WSC}(CodeGen(A), I) \sim out_{FSM}(A, I)$

(2) If $X \sim Y$, then either X and Y are both empty, or

  (a) the first elements of X and Y are equal, and

  (b) for the remaining sequences X' and Y', $X' \sim Y'$ holds.

```
lemma c2 :
  assumes a1
     : "X = WSC_seqOut_clean ( FA_CodeGen C ) ( i ⇝ I )"
  shows "X = Leaf ( Some ( takeStepIO C i ) )
         ⇝ WSC_seqOut_clean ( FA_CodeGen ( nextConfig C i ) ) I"
proof ( cases "i ∈ set ( map Trigger ( Relevant C ) )" )
assume  "i ∉ set ( map Trigger ( Relevant C ) )"
hence " WSC_seqOut ( FA_CodeGen C ) ( i ⇝ I ) =
    Leaf None ⇝ Leaf None ⇝ Leaf None ⇝
    Leaf ( Some ( takeStepIO C i ) )  ⇝ FollowCodeGen i C I"
proof ...( 29 proof steps omitted ) ...done
thus "X =  Leaf ( Some ( takeStepIO C i ) )
    ⇝ WSC_seqOut_clean ( FA_CodeGen ( nextConfig C i ) ) I" using a1
by ( simp  )  ( unfold WSC_seqOut_clean_def  FollowCodeGen_def ,  auto )
next
assume  "i ∈ set ( map Trigger ( Relevant C ) )"
hence "WSC_seqOut ( FA_CodeGen C ) ( i ⇝ I )
    = Leaf None ⇝ Leaf None ⇝ Leaf None ⇝
    Leaf ( Some ( takeStepIO C i ) )   ⇝  Leaf None ⇝ FollowCodeGen i C I"
proof ...( 38 proof steps omitted ) ...done
thus "X =  Leaf ( Some ( takeStepIO C i ) ) ⇝
    WSC_seqOut_clean ( FA_CodeGen ( nextConfig C i ) ) I" using a1
by ( simp ) ( unfold WSC_seqOut_clean_def  FollowCodeGen_def ,  auto )
qed
```

Figure 5: Some Details of the Correctness Proof in Isabelle/HOL

```
theorem "state_sequenceIO A I = WSC_seqOut_clean ( FA_CodeGen A ) I"
proof -
have "bisimulation ( ⋃ I A .
    { ( state_sequenceIO A I , WSC_seqOut_clean ( FA_CodeGen A ) I ) } )"
apply ( unfold bisimulation_def , rule ballI , simp , ( erule exE )+ )
      ...( 17 proof steps omitted ) ...
    apply ( auto , simp add: c1 , simp add: c2 )
done
thus ?thesis
```

Figure 6: Main Correctness Theorem

For this particular case we now define a relation by

$$X \sim Y :\Longleftrightarrow \exists A\ I.\ \begin{array}{l} X = out_{WSC}(CodeGen(A), I) \\ Y = out_{FSM}(A, I) \end{array}$$

This definition trivially fullfills requirement (1). Note that the definition of a bisimulation relation is an artificial construct for conducting proofs. Thus, it remains to be shown that for any non-empty input sequence:

(a) *The first output symbols from a state machine and its generated WSC code are equal.* This can be easily shown by symbolically executing the first steps of our semantics definiton for both WSC and FSMs.

(b) *For the remaining output sequence pair, we can find a finite state machine A such that its output is the left entry of the pair and its generated WSC code outputs the right entry.* For A, we can choose the follow configuration $Follow_i(A)$, which is the same state machine with a different initial state, namely the state in which the original machine is in after the symbol i has occurred. By proving some basic properties of the semantics of both FSM and WSC, we have been able to show within Isabelle/HOL that the code generated from $Follow_i(A)$ produces as output the original sequence minus the first symbol.

Some details of our Isabelle/HOL proof are shown in Isabelle/HOL syntax in Figures 5 and 6. They are used for the proof of the final theorem from Figure 6. It proves equality of FSM A and generated code *FA_CodeGen A* by generating two state sequences and defining a bisimulation containing them. Note that Isabelle implicitly quantifies over all automata A and all kinds of input I. The preceeding lemma (Figure 5) shows that the cleaned output of the generated code, i.e. output of the program with all empty actions removed, is the same as the the action emitted by the first step of the state machine, `takeStepIO C i`, followed by the output of the code generated from the follow configuration. This is the core of the proof as described in Figure 6.

5. RELATED WORK

Apart from the work on semantics of Statecharts discussed in Section 2, there is more related work on the verification of code generation techniques. Verified code generation for Statecharts is closly related to compiler verification since one can regard such a code generator as a special compiler.

In the area of compiler verification, a two-fold notion of cor-

rectness has been established: One distinguishes between the correctness of the translation algorithm itself and the correctness of its implementation. To ensure the first kind of correctness, the correctness of the algorithm, one needs to verify a given translation algorithm as we have verified the Java code generation algorithm in this paper. For the second kind of correctness, a very promising approach is to use program result checkers [10, 11]. Here one does not verify the code generator itself but only its result. An independent checker takes the source and target program, which would be a Statechart and a Java-like program in the context of this paper, and checks whether they have the same semantics. This may ensure correct code generation for each distinct run of the code generator. This technique has also become known as translation validation [23].

Recently our own work has concentrated on verifying compiler optimizations. In [1], we have verified dead code elimination which is a popular compiler optimization. This work also uses bisimulation to define semantical equivalence of programs. In [2], we introduce a principle to model data dependencies with partial orders in order to ease verification. We hope to reuse this concept to further improve our semantics formalism adequate for transformation verification on Statecharts. Summaries can be found in [8, 9].

It should be noted that languages like the Object Constraint Language (OCL) [19] that are frequently used in the UML context may only be used to formulate certain properties such as invariants and pre- and postconditions of UML specifications. In extension, our approach covers the complete semantics of a (Statechart) specification. Hence it is possible to completely verify code generation instead of validating only certain properties.

6. CONCLUSIONS & FUTURE WORK

In this paper, we have demonstrated that it is feasible to specify and verify the transformation from restricted Statecharts to executable program code within the Isabelle/HOL theorem prover. We have specified the semantics of this restricted set of Statecharts as well as of the target programming langugage. Moreover, we have verified a simple code generation algorithm. For this purpose, we have introduced some basic verification principles like bisimulation and explained how they can be used to verify code generation from UML specifications or transformations on UML specifications themselves.

For our future work, we see a large research potential in two directions. First we want to complete our Statechart and Java language specification, thereby in particular verifying more complex code generation techniques. Secondly, we want to verify transformations on Statecharts themselves.

The completion of the Java semantics should be straightforward since formalizations of the semantics of Java in Isabelle/HOL have become very mature in recent years, see e.g. [15]. The authors of this paper regard the establishment of an adequate Statecharts semantics and its formalization in Isabelle/HOL as the most challenging task. The specification of the code generation technique is also a very complex task, especially when one regards optimizations and parallelism. Another area of future research is the verification of transformations on Statecharts themselves. Formal verification of flattening Statecharts might be an actual task for our very near future work.

To achieve verified code generation in practice, it is not sufficient to only verify the code generation algorithm. The implementation of the algorithm might introduce errors as well, cf. our discussion on compiler verification and checkers in Section 5. There are two different principles to guarantee a correct implementation of the code generation algorithm. One could verify the implementation itself. This seems like a rather tedious task which we believe is not yet feasible for real-life implementations. On the other hand, one can verify a simple program result checker that checks for each run of the code generation mechanism that the generated code is a correct translation of the original Statechart. Such a checker can be much simpler than the original transformation implementation. Hence it is easier to verify. Alternatively, one might even generate such a checker from its specification automatically, also a branch of our ongoing work. Such a checker generator could become part of the Fujaba tool suite. We want to tackle these problems in our future work.

7. REFERENCES

[1] J. O. Blech, L. Gesellensetter, and S. Glesner. Formal Verification of Dead Code Elimination in Isabelle/HOL. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods*, Koblenz, Germany, September 2005. IEEE Computer Society Press.

[2] J. O. Blech, S. Glesner, J. Leitner, and S. Mülling. Optimizing Code Generation from SSA Form: A Comparison Between Two Formal Correctness Proofs in Isabelle/HOL. In *Proceedings of the Workshop Compiler Optimization meets Compiler Verification (COCV 2005), 8th European Conferences on Theory and Practice of Software (ETAPS 2005)*, Edinburgh, UK, April 2005. Elsevier, Electronic Notes in Theoretical Computer Science (ENTCS).

[3] M. Born, E. Holz, and O. Kath. *Softwareentwicklung mit UML2 2*. Addison-Wesley, 2004.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[5] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24(2):211–237, 1987.

[6] H. Frank and J. Eder. Equivalence transformation on statecharts. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering, SEKE 2000*, pages 150–158, 2000.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] S. Glesner. Verification of Optimizing Compilers, 2004. Habilitationsschrift, Universität Karlsruhe.

[9] S. Glesner and J. O. Blech. Logische und softwaretechnische Herausforderungen bei der Verifikation optimierender Compiler. In *Proceedings der Software Engineering 2005 Tagung (SE 2005)*. Lecture Notes in Informatics, März 2005.

[10] S. Glesner, G. Goos, and W. Zimmermann. Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers). *it - Information Technology*, 46:265–276, 2004. Print ISSN: 1611-2776.

[11] W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler Correctness and Implementation Verification: The Verifix Approach. In P. Fritzson, editor, *Poster Session of CC'96*. IDA Technical Report LiTH-IDA-R-96-12, Linkoeping, Sweden, 1996.

[12] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[13] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering Methodology*, 5(4):293–333, 1996.

[14] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin*, 67:222–259, 1997.

[15] G. Klein and T. Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298:583–626, 2003.

[16] J. Leitner. Coalgebraic Methods in the Verification of Optimizing Program Transformations Using Theorem Provers. Minor Thesis (*Studienarbeit*), University of Karlsruhe, 2005.

[17] G. Lüttgen, M. von der Beeck, and R. Cleaveland. A compositional approach to statecharts semantics. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 120–129, New York, NY, USA, 2000. ACM Press.

[18] A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of statecharts. In *International Conference on Concurrency Theory*, pages 687–702, 1996. Springer-Verlag, LNCS.

[19] O. Object Management Group. OMG Unified Modeling Language Specification, March 2003. Version 1.5.

[20] O. Object Management Group. UML standard 2.0, 2005. available at http://www.uml.org.

[21] L. C. Paulson. A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions, 2004. available at www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2004/doc/ind-defs.pdf.

[22] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *TACS '91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 244–264, London, UK, 1991. Springer-Verlag.

[23] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, Lisbon, Portugal, April 1998. Springer Verlag, Lecture Notes in Computer Science, Vol. 1384.

[24] A. C. Uselton and S. A. Smolka. A compositional semantics for statecharts using labeled transition systems. In *CONCUR '94: Proceedings of the Concurrency Theory*, pages 2–17, London, UK, 1994. Springer-Verlag, LNCS.

[25] M. von der Beeck. A comparison of statecharts variants. In *Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag.

[26] A. Wasowski. On efficient program synthesis from statecharts. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 163–170, New York, NY, USA, 2003. ACM Press.

[27] A. Zündorf. Rigorous Object Oriented Software Development with Fujaba. Unpublished draft, 2002.

Template- and modelbased code generation for MDA-Tools

Leif Geiger
SE, Universität Kassel
Wilhelmshöher Allee 73
34121 Kassel
leif.geiger@uni-kassel.de

Christian Schneider
SE, Universität Kassel
Wilhelmshöher Allee 73
34121 Kassel
cschneid@uni-kassel.de

Carsten Reckord
SE, Universität Kassel
Wilhelmshöher Allee 73
34121 Kassel
creckord@uni-kassel.de

ABSTRACT

The Model Driven Architecture (MDA) proposes model transformations to obtain an executable model from a platform independent model. Unless one uses an interpreter the common executable model of an application is specified in some programming language. To obtain such an implementation of a model automatically is the task of *code generation* in MDA-Tools. In this paper we present a modelbased approach to this task. It uses explicitly modelled intermediate data and makes use of code templates for the final transformation into pieces of text.

1. INTRODUCTION

CASE-Tools which implement operational semantics do commonly provide either an interpreter or a code generation component to make use of this semantics. In this paper we discuss a concept for such code generation component. The general task of code generation is to transform an abstract syntax graph (ASG) into one or more programming language files. These are compiled (if applicable) and executed to operationalize the specification in the CASE-Tool afterwards.

Our approach to code generation in Fujaba [2] uses Velocity Templates [1] to generate source code in the final step. See Figure 7 for example template code.

To choose the templates to be applied and to supply the template instantiation with parameters an intermediate layer of tokens was introduced. These tokens are created by analysing the ASG elements, for which code should be generated. This enables sorting, optimizations and extensions to work with explicit object structures without altering the ASG of the specification.

1.1 Example

As a simple example we want to show a part of the code generation for a simple graph transformation rule throughout this paper. Generating code for Fujaba's graph transformation rules is one of the core requirements that must be fulfilled by a code generation for Fujaba. The mapping from graph transformations to java code in general is described in [7].

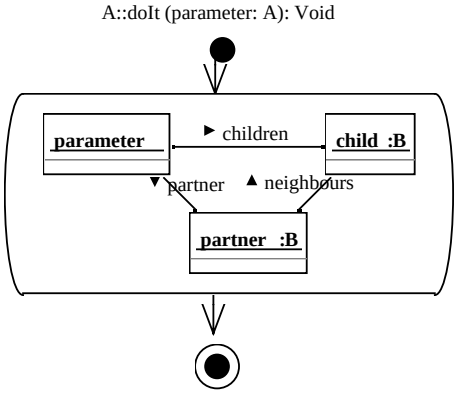

Figure 1: Fujaba rule diagram as an example

The example can be described as follows (cf. Figure 1): The object `parameter` is passed to the rule as method parameter, the `child` object can be found over a link called `children`. Additionally an object named `partner` exists. This can be found by navigating along the link `neighbours` from object `child`. Alternatively it can be found over the `partner` link starting at `parameter`. In this case the object `child` can be found by navigating along the `neighbours` from object `partner`. The rule does not change the object graph (graph-theoretically spoken: RHS equals LHS).

2. CONCEPT

The code generation was split into several subtasks which will be described in detail in the following subsections 2.1 to 2.4. A brief overview is given by Figure 2.

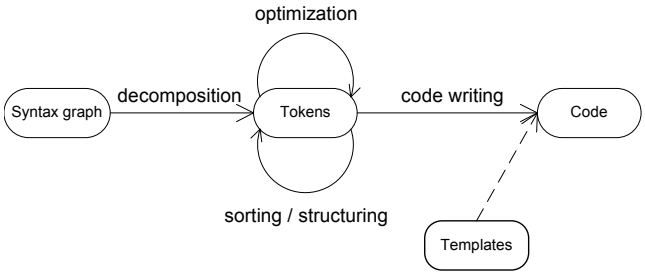

Figure 2: Subtasks of the code generation with initial and resulting data

2.1 Creating Tokens (decomposition)

The first task is to create atomic operation tokens for each syntax-graph element. The same kind of syntax-graph elements can cause different tokens to be created, because of their different attribute values or context. Additionally a single syntax-graph element usually results in multiple tokens. Each token represents an code fragment that should be generated.

The result of the token creation task is a set of tokens which are usually referring each other in several ways (so forming a graph of tokens).

Spoken in terms of our example a token of type `CheckBoundOperation` is created to check if the object `parameter` is bound. For each link a token of type `CheckLinkOperation` is created for each direction the link can be traversed. Figure 3 shows the two resulting `CheckLinkOperations` for the `child` link (the direction is defined by the `subject` link). Tokens for the other two links are created accordingly (not shown). As the `bound` attribute (not displayed in Figure 3) of the `child` object is false, the generated code must search for the object. Therefore a token of type `SearchOperation` is created for each link leading to the object. In addition to the `SearchOperation` along the `child` link shown in Figure 3, another `SearchOperation` for the `child` object is created for the `neighbours` link and two more for the two links connected to the `partner` object. There is no `SearchOperation` for the `parameter` object as it is bound.

Most of the operations require one or more of the involved objects to be matched before they can be performed. These prerequisite objects are specified with `needs` links from the tokens. For example, `CheckLinkOperation t3` needs Objects `o1` and `o2` to check the link between them.

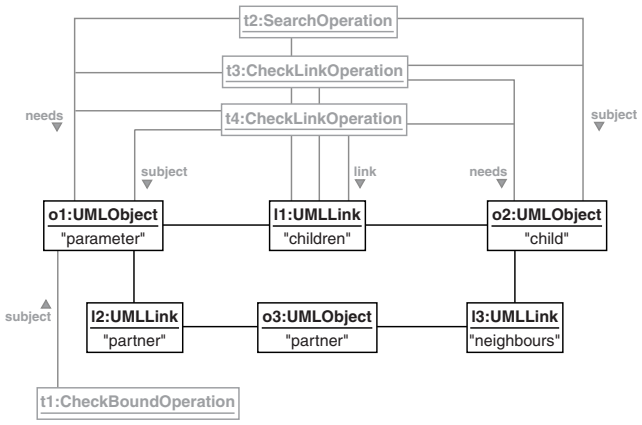

Figure 3: Exemplary tokens (grey) for the transformation rule seen in Figure 1

2.2 Sorting and Structuring

In most cases translating the generated token graph into code directly is difficult. It is easier to first sort and structure the tokens to get a token graph that better reflects possible operational dependencies among tokens. The transformations necessary to structure the tokens depend on the kind of syntax graph the tokens are derived from. Tokens from class diagrams usually need little to no further structuring. Behavioral diagrams on the other hand usually require the token graph to be structured and brought into a hierarchical form that is later mirrored in the hierarchical block structure of the generated code.

We will focus on the structuring of Fujaba's rule diagrams, but similar transformations can be used to structure the token graphs of other behavioral diagrams. A detailed explanation of rule diagrams used in Fujaba can be found in [7]. To generate code for a rule diagram, a *search plan* - an operational form of the diagram - has to be found, that defines how to match the LHS of the rule and how to perform the graph transformation. Since in general many valid search plans exist for a rule diagram, it is also important to find an efficient one, which will be discussed in section 2.3.

In [7] Zündorf describes a basic method to find a search plan and create code directly from the rule diagram. We will now present a method to find a search plan through transformation of the token graph and will then use the found search plan for the template based code generation.

One problem in finding a valid search plan is to decide which tokens are to be used, because usually not all of the generated tokens are needed in a search plan. For example, the objects `child` and `partner` in the example of Figure 1 can both be reached by a `SearchOperation` directly from `parameter`. In that case a `SearchOperation` between `child` and `partner` is not needed and instead a `CheckLinkOperation` can be performed for the `neighbours` link.

The other problem is to sort the used tokens correctly such that all prerequisites of an operation are already matched when the operation is to be performed. In our example an alternative search plan could reach the `partner` object by first finding the `child` from `parameter` and then matching `partner` from `child` over the `neighbours` edge.

Our search plan is a tree structure with ordered child lists, which will be interpreted in a depth-first manner in the rest of the code generation process. Tokens depending on other tokens due to prerequisite objects are located in the subtree below the tokens they depend on, ensuring that all prerequisites are matched when using depth-first traversal.

Transforming the token graph into this search tree is fairly simple:

1. Add a new temporary root node to the LHS graph of the rule diagram and connect all bound objects of the LHS to it. Find a spanning tree starting at this node (using the links as edges).
2. For all edges in the spanning tree the `SearchOperation` towards the child is added to the search tree. Its parent is the `SearchOperation` that finds its sole prerequisite object or the root node if the prerequisite object is bound (cf. Figure 3: `t2` finds `o2` and has `o1` as its prerequisite).
3. For all links not in the spanning tree a `CheckLinkOperation` will be added in the next step. Discard

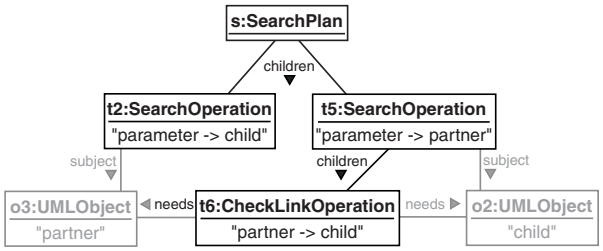

Figure 4: Search plan with partially added CheckOperation

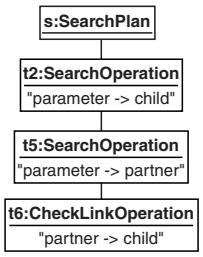

Figure 5: Final Search plan for our example

all other `Search-` and `CheckLinkOperations` (cf. Figure 3: Only one of the three operations `t2-t4` for the `children` link will be in the final search plan).

4. Successively add all `CheckOperations` to the search tree as follows:
   (a) Add the `CheckOperation` as child to a `SearchOperation` that matches one of its prerequisites
   (b) Find a `SearchOperation` for another of its prerequisites. Find the first common ancestor of the two `SearchOperations` and move the subtree with the `CheckOperation` from the common ancestor to the new `SearchOperation`. This is possible because siblings in the tree are independent of each other.
   (c) Repeat for all prerequisites.
5. Add the tokens for the RHS as children of the root node, adhering to the order of object and link destruction, creation and collaborations

The search tree to match all unbound objects in our example consists of two `SearchOperations` matching the `child` and `partner` object. Additionally a `CheckLinkOperation` for the remaining link not used for the search is required. Figure 4 shows the search tree with the `CheckLinkOperation` for the `neighbours` link added below the first of the two `SearchOperations` for its prerequisites as described in step 4.1.

Now the tree has to be modified to get the `CheckLinkOperation` below the other `SearchOperation`, too, as described in step 4.2. Therefore the subtree starting at `SearchOperation t5` and containing the `CheckLinkOperation` is moved below `SearchOperation t5`. Since the `CheckLinkOperation` has no further prerequisites it is now correctly added to the search plan. The resulting, final search plan is shown in Figure 5.

To support easy extensibility, the creation of the token tree is realized with a handler chain similar to the chain of responsibility pattern. The search plan is successively built by the handlers in the chain. The first handler receives the set of available tokens and the empty root node of the tree to be built. Each following handler receives the remaining unused tokens and the tree from the previous handler, restructures or incorporating new tokens into the tree and passes it on. This way, handling of new tokens can be added fairly easy, even though in most cases this will not be necessary because generic handlers exist that can deal with most tokens based solely on their prerequisites, which is usually sufficient.

2.3 Optimization

An important quality feature of the generated code is its runtime efficiency. Therefore we want to find, among the valid search plans, the one that results in the best runtime cost.

The most optimization potential can be leveraged from the selection of the `SearchOperations` used in the initial search tree. Obviously following a to-one link is cheaper than checking multiple objects via a to-many link. Given a cost model for the tokens, a good solution can therefore be found easily by finding a minimal spanning tree to build the initial search tree.

Additionally, fast operations (like link checks) should be performed as early as their prerequisites allow to find invalid matches early and thus avoid further expensive searches. Therefore, when moving a subtree as described in the tree generation process above, its tokens should afterwards be propagated towards the common ancestor as far as their prerequisites allow or until only cheaper tokens are on their path to the common ancestor.

Finally, with the exception of tokens from the RHS of the graph, siblings in the tree are independent from each other. Therefore, subtrees with a low runtime cost relative to the number of tokens in the subtree can be moved to the front of the ordered child lists, again allowing for earlier detection of invalid matches at a lower total cost.

All optimization steps can be easily realized as handlers in the handler chain. The cost model for the tokens is realized as a separate chain of responsibility that can be accessed by all the handlers. For link operations an additional model for the average payload of the referenced link is maintained, separating access costs for the different link types (sorted, ordered, hashed etc.) from the typical number of objects reachable by the link.

In the current implementation the cost and payload models give a static cost estimation, only. They can however be easily extended to e.g. take statistical information gathered from execution on typical data into account.

2.4 Code writing

After having optimized the set of tokens, we are finally able to generate code for them The class responsible for this is

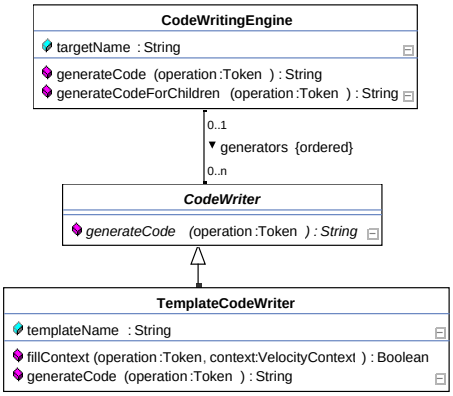

Figure 6: Class diagram for code writing

called `CodeWritingEngine`. It has a list of `CodeWriters` which implements the chain of responsibility design pattern to allow extension, cf. figure 6.

The token tree is visited inorder. Every visited token for which we want to generate code is then passed to the chain, so that the code writer responsible can generate code for the token. This is usually done by an instance of class `TemplateCodeWriter`. This code writer opens the velocity template with the name specified by its `templateName` attribute and passes the token and additionally needed information as context to the velocity template engine. This additional information also includes the code generated for all the children of the token in the tokens hierarchy. Then the velocity engine is used to generate the code.

If e.g. a token of type `ObjectAssignmentOperation` is visited, it is passed to the chain. The object of type `ObjectLifecycleCodeWriter` is responsible for such tokens, so it will initialize the template which is shown in Figure 7. The `ObjectLifecycleCodeWriter` will look up the `UMLObject` which is refered by the token and pass it as `object` parameter when executing the template. In the template in lines 1 to 3 some local variables are set (the name and type of the object and whether or not it is optional. In lines 4-6 the `$tmpName` variable is set depending on whether a type cast is needed or not. In line 9 the object is finally bound. The following lines preform a type check if a type cast is needed.

3. MODELBASED TESTING

Testing the correctness of generated code is generally a hard task. To test code generation, one would start with an arbitrary syntax element in some context and generate code for it. Just comparing the generated code with the expected one would not give a good test criteria: if the code is indented a different way or somehow refactored (different code but same semantics), a test failure would nonetheless be reported. We made the experience that in this case the developer tends to believe that the new code is correct and just overwrites the expected code with the code generated by his new code generation. This way the test would of course execute successful again but possible bugs would have been ignored.

It would be more useful if one could test whether or not the generated code has the expected behaviour. For code

```
1 #set( $name = $object.ObjectName )
2 #set( $type = $object.ObjectType )
3 #set( $optional = $object.isOptional() )
4 #if ( $typeCast )
5   #set( $tmpName = fujaba__TmpObject )
6 #else
7   #set( $tmpName = $name )
8 #end
9 $tmpName = $source ;
10 #if ( $typeCast )
11    ensure correct type and really bound
12   #if ( $optional )
13     if ( $tmpName )
14     {
15        $name = ($type) $tmpName ;
16     }
17   #else
18     JavaSDM.ensure ( $tmpName instanceof $type ) ;
19     $name = ($type) $tmpName ;
20   #end
21 #end
```

Figure 7: Example template that generates Java code for binding an object

which is compiled afterwards, like e.g. our java code, a first hint whether or not the code may be correct is given by the compiler. If the compiler quits with errors, the code is not correct. But obviously this is not a sufficent test criteria.

Our idea is then to run the generated code and test the results. We do this at model level using bootstrapping. To test our java code generation, we use the following approach:

- Structural code, like class definitions, method and attribute declaration, is tested by hand written JUnit tests. Code generation for class definitions for instance is tested using the java compiler, for the most part, which is invoked by a unit test.
- Code generation for method calls within activities is tested by hand written tests as well. In this simple case this is done by comparing the code with the expected one.
- Additional syntax elements of Fujaba's rule diagrams are tested using the modelbased testing approach described in the following paragraph.

The idea of the modelbased testing approach is to model JUnit tests in Fujaba. For these tests code is generated using the code generation to be tested. The tests are then executed using the JUnit framework. The tests should check the behaviour of the generated code by using just the axioms already tested by the hand written tests described above. In more detail, this is done the following way:

- A test class extending the `TestCase` class, provided by the JUnit framework, is modeled in Fujaba.
- Within this class, a unit test, which checks whether or not constraints are interpreted the correct way, can be modeled. This test makes use of method calls, only, which are already tested.

- Assuming that the code generation for constraints does work, what means that the previous test executes successfully, new tests can be modeled which make use of constraints. Such test are tests for the activity diagram parts (sequences, loops, branches).
- On top of this, tests can be modeled, which check additional constructs (creation of objects and links, checking of links, destruction of objects and links...).
- Code for the test modeled in Fujaba is generated using the new code generation and the JUnit tests are executed.

Figure 8 shows the method body for the test method which checks the code generation for to-one link checks. In the first activity two objects are created. The next activity should (if code generation works) check that there is a link between these two objects. If a link is found by the generated code, this is obviously not the desired behaviour (as there is no link between these objects) and a JUnit failure is reported. Otherwise such a link is created and checked for again. If this executes successfully, the generated code has the desired behaviour. The test finishes successfully. That means, if certain parts of the code generation (creation of objects and links as well as sequences of activities and branching) do work, the test in figure 8 checks whether or not code generation for to-one link checking works.

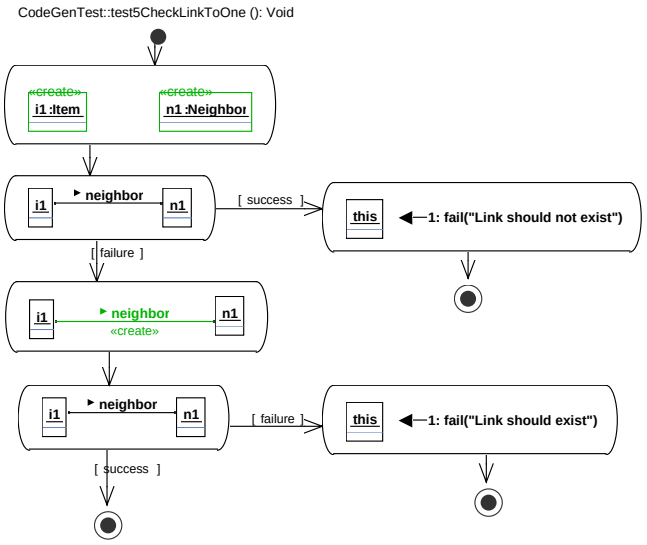

Figure 8: Test method for to-one link checks

4. BOOTSTRAPPING

As Fujaba offers a full-featuered model transforamtion language, it would be a good proove of concept if we model Fujaba with Fujaba itself. Such process is called bootstrapping.

As all parts described above are modeled in Fujaba, bootstrapping Fujaba is finally possible at least for Fujaba's code generation. Till now, not all features of Fujaba are implemented within the new code generation (e.g. support for multi links is still missing). So the bootstrapping process is not yet complete. As soon as we have added these missing features, it would be possible to generate code for the specification in Fujaba using a code generation which was generated by Fujaba itself. This way, it should be possible for a code generation to generate its own code. This bootstrapping is planed for near future using the following process:

If a complete version of the code generation is available:

- Generate code for the new version using the previous code generation.
- Execute the JUnit tests as described in chapter 3.
- Generate code for the new version using itself.
- Execute the JUnit tests against this code generation.
- Generate code from the specification again to ensure, that the generated code equals the one generated before.

5. RELATED WORK

Zündorf describes in [6] how the graph transformations of PROGRES [3] can be split into operations in an operation graph. Then he discusses how to find a search plan (a sequence of search operations) in the operation graph. The search plan is optimized using a given cost model. The decomposition described in chapter 2.1 as well as the optimization in chapter 2.3 uses similar techniques.

In [5] Varró et al. describe a method to find cost efficient search plans from statistical data gathered on typical instance models at design time. Then they propose an adaptive approach that generates multiple search plans and selects the best one at runtime based on statistical evaluation of the current instance model. This approach could be easily incorporated into our current approach since the cost model is well-prepared for more elaborate analysis and the statistical data could easily be gathered by preparing the velocity templates accordingly.

In [7] the transformation from Fujaba's rule diagrams to java code is described. The proposed java code is the basis for our templates discussed in chapter 2.4. A short algorithm for code generation is also stated. Our approach uses a more elaborated algorithm since the algorithm in [7] does not create an intermediate model and only applies few optimization strategies.

The MoTMoT approach [4] also uses a template-based approach to generate code from transformations specified in Fujaba's model transformation language. But unlike Fujaba, MoTMoT does not offer an editor to create story diagrams, but provides a UML 1.4 profile which uses annotated UML class diagrams and annotated UML activity diagrams to model rule diagrams. This way, story diagrams can be drawn with every UML 1.4 compliant editor, like Together, MagicDraw or Poseidon. However, the MoTMoT approach also lacks an intermediate model and elaborated optimization techniques.

6. CONCLUSIONS AND FUTURE WORK

The model-based approach to code generation described in this paper has shown to be very flexible, easy to implement and simple to use. We managed to avoid dependencies to the target textual language in the generator model. All target language elements are expressed in the templates. Only the basic language paradigm (imperative) and some structural information (class, method, declaration hierarchy) is implicitly contained in the implementation.

We expect, introducing new imperative output languages will be possible very quickly. However, this causes creation of multiple similar template files. This tends to increase maintenance cost as behavioral changes in a template must be reflected for all generated langauges. In opposition to that the amount of template code is very low for a single language, compared to the code that was neccessary in the previous Fujaba code generation (more than factor 3).

As the complete code generation model (without templates) is modelled with Fujaba itself this approach paves the way to bootstrapping Fujaba - generating Fujaba with Fujaba. But as well as completing the code generation to support all syntax elements of Fujaba, bootstrapping is still future work.

From the optimizations described in section 2.3, only the minimal spanning tree approach is currently in use. The other methods remain to be implemented. Another area of future work is the optimization based on statistical execution data.

We expect, that the currently implemented transformations, that are used to generate and alter the intermediate data, can be inverted quite easily (except for omitted tokens). This makes us confident that reverse engineering of the generated code to obtain the original model again should be achieved with low cost. Singly the inversion of templates still requires some research work.

7. REFERENCES

[1] Velocity Homepage. http://jakarta.apache.org/velocity/, 1999.

[2] Fujaba Group. The Fujaba Toolsuite. http://www.fujaba.de/, 1999.

[3] Progres Group. PROGRES: Programmed Graph Rewriting System. http://www-i3.informatik.rwth-aachen.de/research/projects/progres/, 2004.

[4] H. Schippers, P. V. Gorp, and D. Janssens. Leveraging UML Profiles to generate Plugins from Visual Model Transformations. In *Software Evolution through Transformations 2003 (SETra03)*. ICGT, Rome (Italy), October 2004.

[5] G. Varró, K. Friedl, and D. Varró. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In *International Workshop on Graph and Model Transformation (GraMoT)*. GPCE 2005, Tallin (Estonia), September 2005.

[6] A. Zündorf. Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme - Spezifikation, Implementierung und Verwendung, PhD Thesis (in German), 1995.

[7] A. Zündorf. Rigorous Object Oriented Software Development, Habilitation Thesis, 2001.

Generation of Type Safe Association Implementations

Dietrich Travkin, Matthias Meyer
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
[travkin|mm]@uni-paderborn.de

ABSTRACT

Model driven development facilitates the specification of software models from which code can be generated automatically. In practice, a software system can often not be modelled completely. Developers still have to implement parts of it manually and thus have to work with the generated code. Therefore, the usability of the code is important.

The Fujaba Tool Suite is a UML case tool which allows to model the structure and behaviour of a system with UML diagrams and to generate Java code for the specifications. However, the code currently generated for associations is not type safe. Furthermore, a lot of code is added to the implementation of the model classes which decreases their usability. In this paper, we present an approach to generate type safe association implementations in Java which also improves the readability and usability of the generated code.

1. INTRODUCTION

Model driven development helps to cope with the continuously increasing complexity of the software systems being developed today. Instead of implementing a whole system by hand in some programming language, models are designed in modelling languages, e.g. UML. Then, code can be generated automatically from these models. Today, typically not the whole system can be modelled and generated, so that developers still have to work with the generated code manually.

When modelling a system with UML, its structure is defined with class diagrams. Class diagrams allow specifying various kinds of associations between classes. However, today's programming languages have no support for associations. Thus, when generating code for class diagrams, associations have to be expressed using the language constructs available in the particular programming language.

The Fujaba Tool Suite [3] is a UML case tool that offers UML class diagrams to model the structure and extended UML activity diagrams, called story diagrams, to model the behaviour of a software system. Fujaba's class diagrams support most binary association types defined by the UML (qualified, sorted or ordered, 1-to-1, 1-to-n, n-to-m, bi- and unidirectional).

Furthermore, Fujaba is able to generate Java code for its models. Classes modelled in class diagrams are translated to Java classes. To implement the associations, the classes participating in an association get additional private attributes to store associated objects and public access methods to link, unlink, or iterate through associated objects. The access methods also ensure consistency for bidirectional associations by mutually calling each other.

Depending on the type of association, up to 18 access methods per association are generated into a model class. Thus, a class participating in several associations becomes badly readable. Furthermore, the generated code is not type safe. Thomas Maier and Albert Zündorf [5, 4] recognised these problems and propose a different solution. They greatly improve the readability and usability of the generated code by extracting the association methods into separate classes representing association ends, so called role classes. However, their approach can not guarantee type safety as well and the support for associations which are qualified on both sides remains unclear.

In this paper, we present an approach that adopts the idea presented in [5]. We also implement associations using general role classes which contain all necessary access methods. However, our hierarchy of role classes is organised differently to explicitly support associations which are qualified on both sides. Furthermore, by generating specialisations of role classes for each concrete association end, we are able to offer a type safe implementation. The role class hierarchy is available as a separate class library which can be used independently of Fujaba. In addition, a Fujaba plug-in has been developed which adapts the code generation of Fujaba for non-qualified associations to use the new approach.

In the following section we present requirements for an association implementation and point out the limitations of the existing approaches in more detail. Section 3 describes our approach for implementing associations and Section 4 describes a Fujaba plug-in which generates code according to it. Section 5 concludes the paper and indicates future work.

2. REQUIREMENTS

There are many ways to implement associations. In all cases the implementations have to manage the references between the objects connected by an association depending on its kind (e.g. qualified 1-to-n) and its constraints (e.g. ordered). In [6] we identified several requirements which should be fulfilled by an association implementation. The four most important of them which we address in this paper are:

1. **Association types**
   Associations may be uni- or bidirectional with multiplicities 1-to-1, 1-to-n or n-to-m. In case of to-n as-

sociations, the associated objects may be ordered or sorted. Furthermore, associations may be qualified on one or both sides. All combinations have to be supported.

2. **Consistency**
   In case of bidirectional associations, consistency has to be ensured: if there is a reference from object a to object b then there also has to be a reference from object b to object a. Thus, when object a is linked/unlinked to b, the reverse link from b to a has to be established/removed automatically.

3. **Type safety**
   The generated code has to be type safe, i.e. all type errors have to be detected at compile time.

4. **Readability**
   The generated code has to be human readable and should add as little code as possible to model classes.

The current Fujaba version fulfills the first two requirements. For each association, private attributes with several public access methods are generated into the code for the model classes. The access methods allow to link or unlink instances of model classes according to the association. In case of bidirectional associations, these access methods automatically ensure consistency by calling each other. When an access method is called on object a to connect it to object b, the method calls the corresponding access method on b to link it to a. The same holds for the access methods to unlink objects. Thus, to establish or remove a link, a call of the appropriate access method on one of the objects is sufficient.

For the implementation of to-n associations, special container classes are used to store an arbitrary amount of associated objects. These containers allow storing objects of the most general type (Object in Java) and the generated code contains type casts. Thus, the code is not type safe and requirement 3 is not fulfilled. A developer is able to (accidently) insert code into a model class that adds objects of the wrong type to a container managing the connected objects of an association. Since this can not be checked by the compiler, type errors occur at runtime.

Depending on the type of association, up to 18 public access methods per association are generated into the implementation of a model class. This heavily decreases the readability and usability of the generated code and the public interface of the model classes becomes very bloated. Requirement 4 is not fulfilled.

The approach presented in [5] greatly improves the readability of the generated code by providing the functionality to manage the associated model elements in separate role classes and thus keeping the code added to the model classes at a minimum. The role classes are implemented using Java Generics. Generics [1, 2] are new with Java 1.5 and enable generic type definitions in Java. However, in spite of using Generics, the resulting association implementations are not type safe. In order to ensure consistency for bidirectional associations, inside the role classes Java's reflection mechanism is used to call methods on the opposite end of the association. The reflection mechanism requires type casts and thus type safety is lost (cf. [6] for details). In addition, it is unclear how associations which are qualified on both sides are supported. Thus, requirement 4 is additionally fulfilled but requirement 3 is not.

3. TYPE SAFE ASSOCIATIONS

In the following, we present an approach which fulfills all the requirements stated above.

3.1 Implementations based on Role Classes

We adopt the idea proposed in [5] to separate the association implementation and the model implementation from each other. Instead of generating all the code to manage references of associated objects into the model classes, the functionality is provided by separate role classes. The role classes contain all the functionality to link or unlink two model elements and to iterate through the connected elements.

For each kind of association end, a special role class exists implementing the required association methods. In case of a 1-to-1 association, each model class must be able to reference one instance of the other model class at runtime. Thus, a to-1 role class is required which is able to manage one reference. In case of a 1-to-n association, one class must be able to manage an arbitrary number of references to instances of the other class, requiring a to-n role class. N-to-m associations can be realised with the help of two instances of a to-n role class. Association constraints and qualified associations require additional role classes.

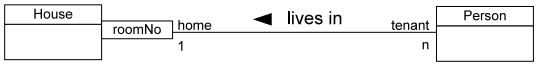

Figure 1: A qualified bidirectional 1-to-n association

```
class Person
{
   private Tenant_Home_Role home = null;
   public final Tenant_Home_Role home()
   {
      if (this.home == null)
      {
         this.home = new Tenant_Home_Role (this);
      }
      return this.home;
   }
}
```

Figure 2: An implementation of a model class

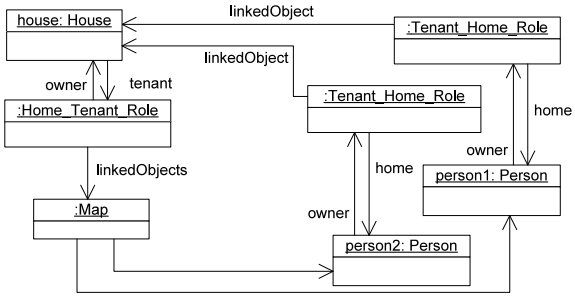

Figure 3: Realisation of the association in Figure 1

An association is realised by two instances of role classes, one for each association end. Instead of the association

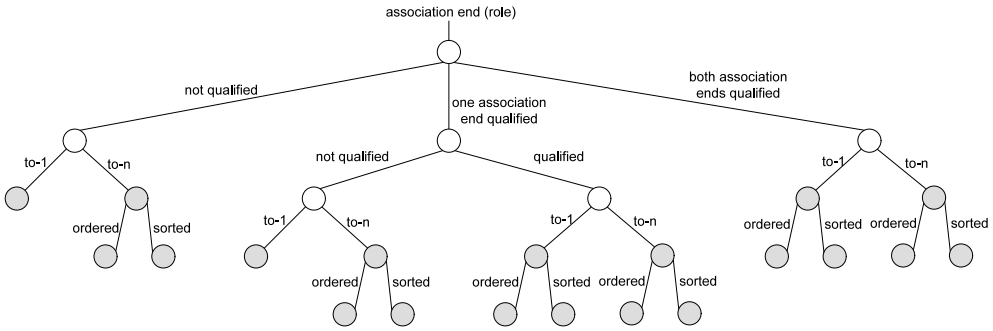

Figure 4: A decision tree describing the role class hierarchy

methods the model classes only get one attribute and one method for each association they are involved in. The attribute saves the role object and the method creates and returns it. Thus, the association methods are implemented only once in a role class and do not have to be generated for each association anymore, which avoids code redundancy. As an example, Figure 2 illustrates the implementation of a model class involved in the association shown in figure 1. Figure 3 shows an object structure in which a House object is connected with two Person objects via the same association using role objects. Note that the figures already use specialised role classes which will be explained in Section 3.3.

3.2 Role Class Hierarchy

Each of the role classes implements a link and an unlink method. In bidirectional associations, these methods have to call each other on either side of an association to maintain consistency. Qualified associations require a key to link two model elements. Since the methods call each other on both sides of an association, they require this key, regardless whether they are called on an object representing a role that is qualified or not (cf. Figure 5). If both sides are qualified even two keys are needed. Therefore the number of parameters in the link and unlink methods are different depending on whether the role is used in an association that is not qualified, qualified on one side or qualified on both sides. Thus, a common abstraction for all role classes would only be possible, if all link and unlink methods had three parameters (one for the object to be linked or unlinked and two for possible keys). In many cases, however, the additional parameters would be useless. Therefore, we propose a hierarchy of role classes which is divided into three rather independent sub hierarchies for associations which are not qualified, qualified on one side, or qualified on both sides, respectively. Inside these hierarchies, the number of required parameters is equal and a common abstraction exists.

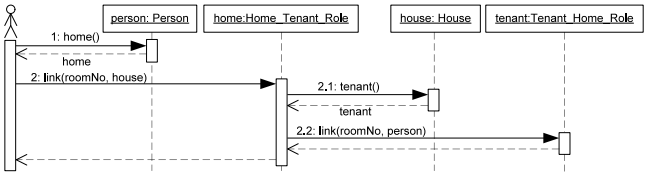

Figure 5: A call of the link method in the qualified association shown in Figure 1

Figure 4 shows a decision tree which describes the whole role class hierarchy and helps to determine the role classes needed to realise a particular association. Each node in the tree corresponds to a role class. An edge is labelled according to the purpose of the child role class and indicates that the child role class refines its parent. At the first level, the tree is divided into the three sub hierarchies. The left hierarchy shows the role classes for non-qualified associations. It is subdivided into a class for the to-1 side and one for the to-n side which again has two children, one for an ordered and one for a sorted to-n side. The hierarchy in the middle contains the role classes for associations qualified on one side. It is subdivided in roles for the non-qualified side and those for the qualified side which in turn are organised according to the type (to-1 or to-n side) as well as to possible constraints (ordered and sorted). The hierarchy on the right shows the roles for associations qualified on both sides and is organised according to the same criteria as the other hierarchies.

Although the number of different role classes is rather high, it is easy to choose the right role classes for a particular association. The decision tree in Figure 4 describes how to do that (dark nodes indicate possible decisions). The association between House and Person (cf. Figure 1), for example, is qualified on one side. Thus, role classes from the hierarchy in the middle have to be used. A House object must be able to manage an arbitrary number of Person objects for each key. Therefore it requires a qualified to-n role. A Person object must be able to store one House object. Since the opposite side is qualified and this side does not use a key, a non-qualified to-1 role class from the hierarchy in the middle has to be chosen.

The role classes are generic. They use type parameters for the type of the elements to be referenced by the roles, the type of the role owner and – in qualified associations – the key types. The generic type definitions are necessary for a type safe implementation but not sufficient. This is described more precisely in the following section.

3.3 Type Safety

To keep bidirectional associations consistent, methods for linking and unlinking two objects are called on both sides of an association (link on one side of an association calls link on the other side of the association, cf. Figure 5). The link and unlink methods are implemented within the general role classes, where the types of the elements referenced by the role classes are only represented by type parameters. The concrete types used in a concrete association are not known

and thus the access method for the role object of a model element (e.g. method home in Figure 2) is not known. This makes a call of the link or unlink method on the other side of an association impossible.

However, the access methods in the model classes can be revealed by subtyping the generic role classes and binding its type parameters to the concrete types of the model elements involved in a concrete association. In each general role class in the hierarchy, an abstract method getOppositeRole is declared which takes a model element to be linked or unlinked as argument. The method is meant to return the role object from the given model element which represents the opposite end of the same association. It is used inside the general role class implementations to get the (opposite) role object from a model element to be linked or unlinked. Thus, all role classes in the hierarchy are abstract. They contain the complete implementation except of the getOppositeRole method.

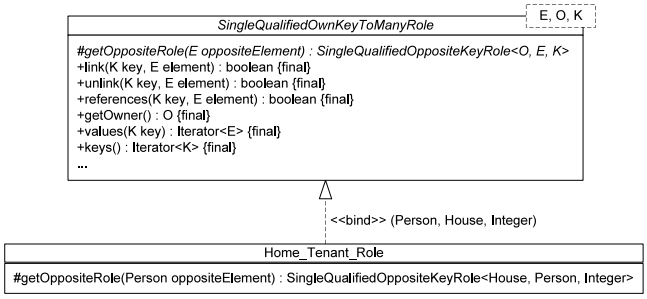

Figure 6: A specialisation of an abstract role class

```
public class Home_Tenant_Role
  extends SingleQualifiedOwnKeyToManyRole<Person,House,Integer>
{
  public Home_Tenant_Role(House owner)
  {
    super("home", owner);
  }
  protected SingleQualifiedOppositeKeyRole<House,Person,Integer>
    getOppositeRole(Person oppositeElement)
  {
    return oppositeElement.home();
  }
}
```

Figure 7: A specialisation of an abstract role class

For each association end, a concrete subclass of a generic role class has to be implemented in which the type parameters are bound to the concrete types of the model elements. Inside these classes, the getOppositeRole method has to be implemented (cf. Figures 6, 7). The concrete subclasses only contain a constructor as well as the getOppositeRole method and thus are very small. Without them, however, a type safe access to the role representing the opposite end of an association would not be possible.

4. CODE GENERATION

The role class hierarchy described in Section 3.2 has been implemented in a class library which can be used to implement associations independently of Fujaba.

In addition, a Fujaba plug-in has been implemented which uses the role class library to generate code for non-qualified associations according to the approach presented in this paper. The plug-in generates the specialised role classes required for each association. To further increase the usability, each concrete role class is placed in a subpackage of the corresponding model class' package named roles. The code generated for model classes uses the specialised role classes to realise the associations. Furthermore, the code generation for story diagrams has been adapted. The code generated for story diagrams calls access methods of associations to create or destroy links between objects.

5. CONCLUSIONS AND FUTURE WORK

This paper describes an adaption and extension of the ideas presented in [5] leading to type safe association implementations with improved readability and usability. The functionality for managing associations is provided by separate role classes and no longer generated into the model classes. By specialising the role classes for each concrete association end the code is made type safe. A class library with all necessary role implementations is available and can be used to implement associations independently of Fujaba. A Fujaba plug-in adapting the code generation of Fujaba for non-qualified associations is available as well.

Fujaba itself is partially implemented using its own code generation mechanism. Re-generating the Fujaba code using the new association implementations would increase its usability. Furthermore, type errors in association implementations would be revealed already at compile time.

The next step could be to support the modelling of generic types with UML templates in Fujaba. This would enable the generation of completely[1] type safe code.

6. REFERENCES

[1] G. Bracha. *Generics in the Java Programming Language*. Online at: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, July 2004, Tutorial.

[2] G. Bracha, N. Cohen, C. Kemper, S. Marx, M. Odersky, S.-E. Panitz, D. Stoutamire, K. Thorup, and P. Wadler. *Adding Generics to the Java Programming Language: Participant Draft Specification*. Online at: http://www.jcp.org/aboutJava/communityprocess/review/jsr014/, April 2001.

[3] Fujaba Development Group. *Fujaba ToolSuite*. Online at: http://www.fujaba.de, 2004.

[4] T. Maier. *Associations*. Online at: http://sourceforge.net/projects/associations/, November 2004. Version 0.4.

[5] T. Maier and A. Zündorf. *Yet Another Association Implementation*. In *Proceedings of the 2nd International Fujaba Days, Darmstadt, Germany*, pages 67–72, Septemeber 2004.

[6] D. Travkin. *Generierung typsicherer Implementierungen für Assoziationen in UML-Modellen (in german)*. Bachelor's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, February 2005.

[1] This paper only focuses on the code generated for associations. The code generated by the described Fujaba plug-in is type safe for associations only.

The SceBaSy PlugIn for the Scenario-Based Synthesis of Real-Time Coordination Patterns for Mechatronic UML*

Holger Giese and Sergej Tissen
Software Engineering Group, University of Paderborn,
Warburger Str. 100, Paderborn, Germany
[hg|serti]@uni-paderborn.de

ABSTRACT

The future generation of networked, technical applications demands support for the development of high quality software for the proper real-time coordination of safety-critical systems. In this paper, we present the SceBaSy plugin for the Fujaba Real-Time Tool Suite which supports the scenario-based synthesis of the real-time coordination patterns. Extending our approach for the compositional formal verification of MECHATRONIC UML models described by components and patterns [5], the plugin enables the designer to specify the required real-time coordination using multiple parameterized scenarios described by a subset of the UML 2.0 sequence diagram notation. In addition to the synthesis, comfortable analysis capabilities have been realized to guide the designer when conflicts between the different scenarios exist.

1. INTRODUCTION

In the development of safety-critical systems, the design and verification of the real-time coordination of the system is of crucial importance. The increasing complexity of these systems and their interconnection by networks which can be often observed today, makes their production an even greater challenge. Therefore, the current practice could benefit from more automated support for the design of correct and safe real-time coordination.

Today, a number of scenarios are usually developed in the earlier phases to outline and specify possible or required interaction behavior. Later, an operational model of this interaction is then derived manually. The underlying idea of scenario-based synthesis is simply to automate this step (cf. [7, 8, 12, 13]).

However, in our specific case of real-time systems, besides the causal relation between the different events also the timing constraints are essential. Available approaches cf. [10, 9, 6] only provide a global behavior for fixed timing constraints. We employ here our approach [4] for the synthesis of distributed operational behavior from parameterized scenarios, as it is in practice often difficult to specify all timing information such as worst-case execution times (wcet), deadlines, or timeouts in advance.

In this paper, the support for the scenario-based synthesis of real-time coordination pattern provided by the SceBaSy plugin is presented. It complements our MECHATRONIC UML approach for the compositional formal verification [5] which supports to build a correct and safe real-time coordination of a whole system composed of components and patterns.

The plugin supports the automatic derivation of the parameterized role behavior in form of Real-Time Statecharts (RTSC) [2] for patterns from a given set of parameterized scenarios. Therefore, it enables the designer to automate this otherwise costly development step and guarantees correctness by construction. In addition to the synthesis, comfortable analysis capabilities have been realized to guide the designer when conflicts between the different scenarios exist.

The paper is organized as follows: In Section 2, the real-time modeling with scenarios as supported by the SceBaSy plugin is sketched introducing a simple example pattern. Then, the support of the plugin for the analysis of a set of scenarios in case of conflicts is outlined in Section 3. For a conflict free set of scenarios, the handling and output of the synthesis step are then described in Section 4. Finally, we sketch the architecture of the plugin and its dependencies w.r.t. other MECHATRONIC UML plugins in Section 5 and sum up the paper with a short conclusion.

2. REAL-TIME MODELING

As a running example, we consider the *Monitor-Actuator Pattern* [3]. This pattern specifies a controller which monitors and controls another system. Therefore, the controller sends advices to the system via an actuator and monitors their realization. The actuator calculates the actions which have to be done to realize the system-state and sends them to the system. The monitor waits for the system status and decides then whether the advice is fulfilled or not. Furthermore the monitor and actuator check their presence by sending periodically a life tick message to each other (cf. heart beat).

To describe the variable behaviors of the roles within the pattern, we model the different scenarios by UML sequence diagrams where besides constant timing constraints also parameterized timing constraints for upper bounds are supported. The sequence diagrams are modeled with the Fujaba plugin UMLSequenceDiagrams. Additionally, our plugin extends the sequence diagram plugin by time observations and time restrictions to be able to specify real-time behavior within sequence diagrams. Time observations assign the actual time to a clock and time constraints can refer to these clocks and make restrictions. Two sequence diagrams which describe two scenarios of the *Monitor-Actuator Pattern* are shown in Figure 1 and 2

*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.

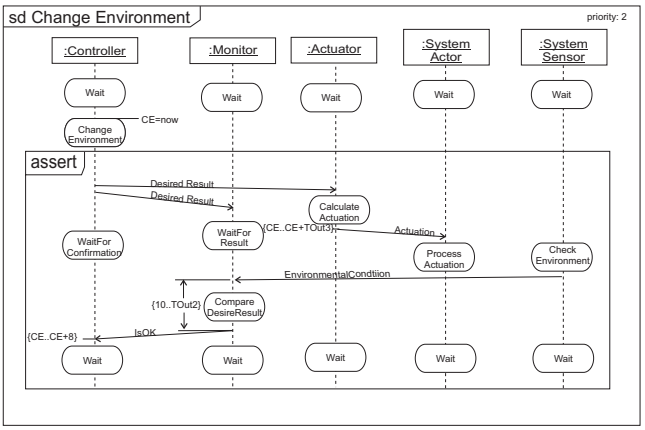

Figure 1: Change Environment

Figure 1 shows the scenario which models the standard system flow. The controller initiates a state change and awaits the result. The controller sends a message to the actuator who then calculates the necessary actions and sends them to the system. Also the monitor receives the same message from the controller to observe the achievement. After receiving the system-state, the monitor compares it with the desired state and sends an okay message to the controller. After at most $TOut_3$ milliseconds (ms), the actuator has to complete his calculations and send his advices to the system. The maximal available ms for the monitor are $TOut_2$. The whole activity must not exceed 8 ms.

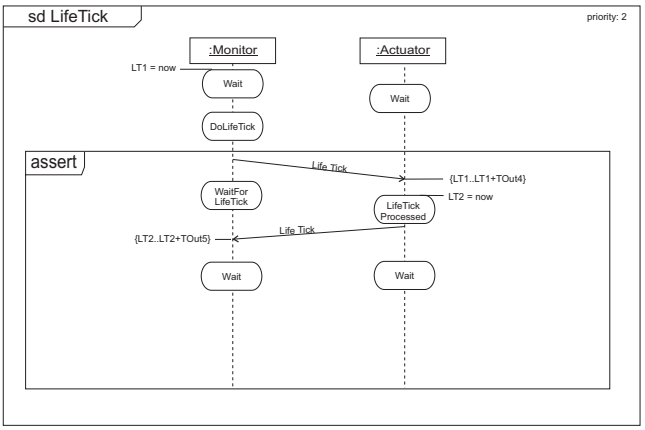

Figure 2: Life Tick

Figure 2 specifies a scenario where the monitor sends a life tick message to the actuator and the actuator responds on his part with a life tick. The monitor has to send a life tick every $TOut_4$ ms and after receiving the life tick the actuator has maximal $TOut_5$ ms to answer.

After we have modeled the scenarios by sequence diagrams, we must create a new synthesis task. When we have entered a name for the task, a new tab named *Synthesis* is generated in the project tree. This panel manages all created scenario-based synthesis-tasks. Figure 3 displays the user interface for synthesis tasks. The root node is the name of the synthesis. The subnode *Sequence Diagrams* holds and manages all sequence diagrams which were imported into the task. The *Settings* node allows setting, removing or modifying additional inequalities which restrict the parameters employed in the sequence diagrams. In addition, we can set weight for all parameters to define which ones should be preferred in contrast to others. The subnode *Pattern* displays the synthesized real-time statecharts and their properties.

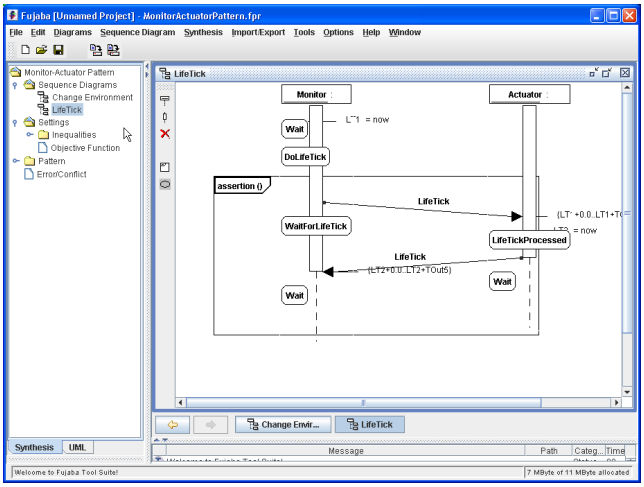

Figure 3: SceBaSy User Interface

3. ANALYSIS

To analyze the sequence diagrams and their timing constraints, our plugin first maps the sequence diagrams to time constraint graphs (TCGs). A TCG represents all possible paths within a sequence diagram and formalizes time observations and constraints. TCG nodes depict possible states of the roles within a sequence diagram, and edges are used to describe how time passes on the lifeline.

In sequence diagrams, *activities* are used to describe the execution of a side-effect. We assume that the specific execution time of an activity is usually unpredictable. However, we can assume lower and upper bounds (cf. worst-case execution times (wcet)) for the activity. The communication in sequence diagrams can be asynchronous or synchronous. Unlike the asynchronous communication in sequence diagrams, TCGs only provide synchronous communication. To address this problem, our plugin generates additional channels which simulate asynchronous communication via buffering.

The plugin maps the timing constraints used in sequence diagrams to constraint edges with a uniquely determined starting point, when the time observation is set, and an end point denoted by the time constraint itself. To reflect the assert blocks within the sequence diagrams, the plugin represents nodes which relate to a state within an assert block in the sequence diagram by assert nodes, all other nodes become possible nodes.

To verify the correct timing behavior of the TCGs, we have to take *consistency* into account which requires that always the same time will elapse on two alternative paths between two nodes. In addition, we demand *locality* [4], which requires that the timing of local tasks only depends on the current state.

To address the problem of *consistency* and *locality*, our plugin derives a set of inequalities which describe the execu-

tion time dependencies [4] and checks their feasibility. This includes all subgraphs of the TCGs which result from time observations and time constraints. Please Refer to [4] for a more precise description. A set of linear inequalities is only feasible iff an assignment for all variables exists which fulfills all inequalities.

If the linear inequality system is not feasible, the integrated conflict handler has to find out which inequalities exclude each other [11]. The conflict handler traces back the inequalities to the affected time constraints or manually set restrictions, and shows an error message to the user. In addition, the constraints in conflict are highlighted by the plugin in the related diagrams.

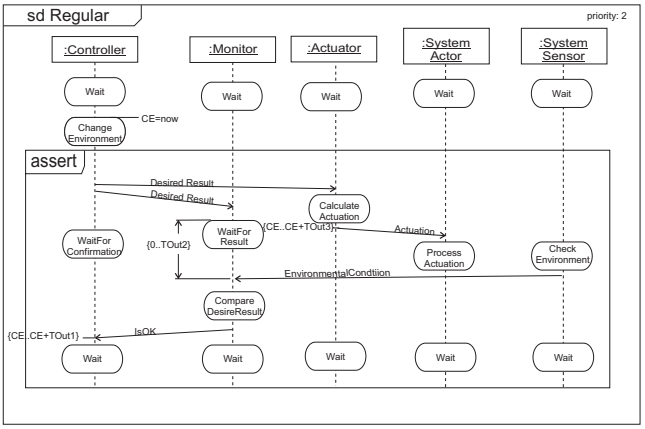

Figure 4: Time Constraint conflict resolved

Figure 4 shows the corrected *Change Environment* scenario where two constraints have been in conflict. The Monitor required min. 10 ms to compare the desired result with the actual environment, in contrast the Controller awaited the result of the calculation within 8 ms (cf. {10 ..TOut2} and {CE ..CE+8} in Figure 1).

In addition to the outlined analysis, asserted blocks have to be taken into account by the inequality system, and we have to check for contradictory state changes in a single state. Please refer to [11] for more details about these and other analysis steps.

4. SYNTHESIS

The previous chapter suggested how sequence diagrams can be analyzed and verified. Now we want to synthesize the behavior for the involved roles. The synthesis-algorithm handles every sequence diagram consecutively. Thus, existent real-time statecharts are extended to the content of a sequence diagram iteratively. One iteration step is as follows: The mapping to states is simply derived for each node using the state labeling of the sequence diagram. Local state changes in sequence diagrams are mapped to transitions between states in a real-time statechart. The communication expressed by a message in a sequence diagram becomes a communication transition in both related statecharts. For the generated transitions, the time conditions simply result in a time guard and deadline such that the specified timing constraints are satisfiable. For more details about mapping sequence diagrams to real-time statecharts refer to [4].

After generating operational behavior for each role, the SceBaSy plugin is able to optimize the real-time statecharts [4]. The optimization algorithm is able to collapse redundant states and their transitions which result from the synthesis into a single state as well as hierarchical states. This optimization can be turned on or off for every pattern role individually. The application of these syntactical rules erases 9 states from the monitor's real-time statechart and results in the model depicted in Figure 5.

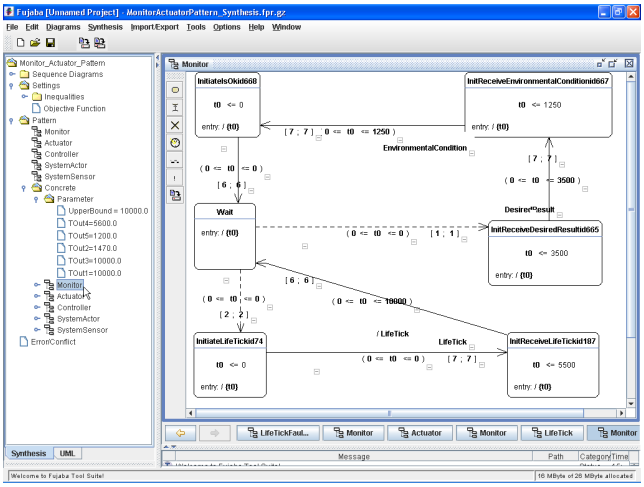

Figure 5: Optimized behavior for the Monitor

Once valid parameters are calculated and real-time statecharts are derived, the plugin uses the model checking capability of the real-time version of Fujaba to ensure that the synthesis result for the given parameter values is free from deadlocks or time stopping deadlocks.[1] After the pattern is successfully model checked, the entire Real-Time Coordination-Pattern is generated as shown in Figure 6. Then each Role gets its own behavior assigned in the form of a real-time statechart.

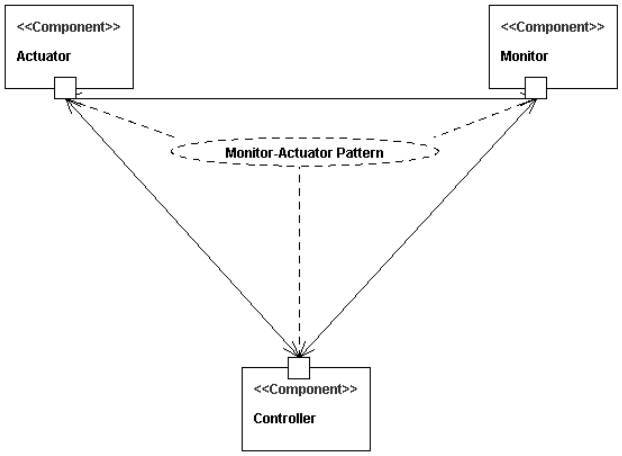

Figure 6: Monitor-Actuator Pattern Structure

[1]It is to be noted, that problems due to time stopping deadlocks and thus reachability can only be proven using a real-time model checker after all parameters have been set, as the emptiness problem for parameterized timed automata with more than 2 parameters is undecidable [1].

The user has now the ability to evaluate the real-time statecharts, to modify the parameter weights, to add or remove inequalities and adapt the sequence diagrams. Now he can initiate the synthesis again to get the readjusted results.

5. PLUGIN

Sequence diagrams used for modeling the scenarios are implemented in a Fujaba Plugin named UMLSequenceDiagrams and the synthesized real-time Statecharts are realized in the plugin RealtimeStatechart. Our plugin extends the UMLSequenceDiagram plugin by the ability of adding timing constraints to sequence diagrams. The plugin UMLRT2 provides the ability to save and restore our generated coordination pattern in a repository.

The SceBaSy plugin provides a standardized interface to solve the linear inequalities. Thus, different inequality solver can be used by the plugin. So far we have made use of two solvers: A java implementation of the simplex algorithm and a commercial package named CPLEX[2].

To evaluate our plugin, we use an extended version of our running example, where the additional sequence diagrams use partially the same parameters in their constraints like the ones presented. We simply add a sequence diagram to the synthesis in every new experiment and record the times and characteristics. Table 1 depicts the results of these experiments. While the simplex package shows a moderate increase in computation time, the commercial CPLEX package does not show an increase in computation time at all. Even though much more experience with the plugin is require to really judge the scalability problem, the experiments are promising.

Number of sequence diagrams	1	2	3	4	5
Numb. of inequalities	54	168	194	220	237
Time in ms to solve the ineq. system (CPLEX)	7	18	21	10	10
Time in ms to solve the ineq. system (Simplex)	19	31	61	69	96
Total runtime [ms]	1170	1930	2056	2671	3102
Numb. of states	13	43	49	55	60
Numb. of transitions	12	44	61	86	95

Table 1: Evaluation data for the SceBaSy plugin

6. CONCLUSIONS

We describe in this paper the SceBaSy plugin for the automatic synthesis of correct real-time coordination patterns from parameterized scenarios. We outlined how the static analysis capabilities of the plugin can be used to analyze problems within a given set of parameterized scenarios. In a next step, the plugin permits to synthesize the real-time behavior for each role of a parameterized real-time pattern in form of parameterized RTSC. In addition, the plugin permits to derive appropriate parameter setting and thus the developer can systematically study the trade-offs between them. When valid parameters for the real-time statecharts have been set, the model checking feature of Fujaba could be used to ensure that the synthesis result is free from deadlocks or time stopping deadlocks.

[2]http://www.ilog.com/products/cplex/

REFERENCES

[1] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 592–601. ACM Press, 1993.

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

[3] B. Douglas. *Doing Hard Time*. Addison-Wesley, 1999.

[4] H. Giese, F. Klein, and S. Burmester. Pattern synthesis from Multiple Scenarios for Parameterized Real-Timed UML models. In S. Leue and T. Systä, editors, *Scenarios: Models, Algorithms and Tools*, volume 3466 of *Lecture Notes in Computer Science (LNCS)*, pages 193–211. Springer Verlag, April 2005.

[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] D. Harel and R. Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proc. 10th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, Fort Worth, Texas, USA, 2002. (invited paper).

[7] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In F. J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[8] E. Mäkinen and T. Systä. MAS - an interactive synthesizer to support behavioral modeling in UML. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Canada*, pages 15–24, May 2001.

[9] A. Salah, R. Dssouli, and G. Lapalme. Implicit integration of scenarios into a reduced timed automaton. *Information and Software Technology*, 45:715–725, August 2003.

[10] S. Somé, R. Dssouli, and J. Vaucher. From Scenarios to Timed Automata: Building Specifications from Users Requirements. In *Proceedings of the 1995 Asia Pacific Software Engineering Conference (APSEC '95)*, 1995.

[11] S. Tissen. Szenario-basierte Synthese für parametrisierte, zeitbehaftete UML Sequenzdiagramme. Bachelor's thesis, University of Paderborn, Software Engineering Group, Paderborn, Germany, April 2005.

[12] S. Uchitel and J. Kramer. A Workbench for Synthesising Behviour models from Scenarios. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Canada*, pages 188–197, May 2001.

[13] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on on Software engineering June 4 - 11, 2000, Limerick Ireland*, 2000.

Worst-Case Execution Time Optimization of Story Patterns for Hard Real-Time Systems*

Sven Burmester†, Holger Giese, Andreas Seibel, and Matthias Tichy
Software Engineering Group, University of Paderborn, Warburger Str. 100, Paderborn, Germany
[burmi|hg|aseibel|mtt]@uni-paderborn.de

ABSTRACT

In the future, technical systems are expected to operate more intelligent than today by taking their local context explored by means of sensors and network communication into account. To realize this vision, the systems must be able to represent and query as well as interact with a large number of possible situations not known a priori. Therefore, flexible means to store, query, and manipulate such context information are required. Known flexible and powerful representations are class diagrams or other graph-like notations. However, such dynamic data structures which are sources for unpredictable run-time timing behavior are traditionally not recommended for the development of hard real-time systems. In this paper, we describe our efforts to employ story patterns, which are used for the specification of query and update operations on dynamic data structures, in hard real-time systems.

Keywords

Real-Time, Story-Pattern, Worst-Case Execution Time Optimization

1. INTRODUCTION

Advanced technical systems of the future such as self-adaptive [18, 13, 14] or self-optimizing [3] technical systems will operate smarter than today's systems by adjusting their operation to the experienced context. Besides the information provided by sensors, the communication with other entities near by via wireless networks will increase the available information and its complexity.

The software of these systems must thus be able to represent and query a large number of possible not a priori known context situations. The means to store, query and manipulate such context situations must support model-based development and should not be restricted to fixed-sized arrays. UML and in particular class diagrams became the standard to describe the structure of the complex information. Story diagrams [19, 12] are an advanced technique to manipulate this information. However, class diagrams describe properties of dynamic data structures which result in unpredictable run-time timing behavior and are thus traditionally not considered as an option for the development of hard real-time systems.

In real-time systems, the provision of a service (e.g. reaction to an incoming message, a computation, ...) is associated with a certain deadline. If the deadline expires before the service is provided, the results in embedded systems are typically catastrophic due to damages to humans in case of automotive or railway systems. Those systems are named *hard real-time systems*.

In order to guarantee to meet the required deadlines, the worst-case execution times (WCETs) of methods or other implementation artifacts must be known. In addition, worst-case execution times are required for a schedulability analysis [5] which is used to check whether concurrent processes are executable on a given processor meeting the required deadlines.

Standard dynamic data structures are unbounded, i.e. they have no predetermined maximal amount of stored elements. Thus, no worst-case execution time can be given for operations on these data structures, since the execution time typically is dependant on the contained number of stored elements. Therefore, in order to determine a worst-case execution time, the maximal number of elements in those data structures must be fixed beforehand. Then, the worst-case execution time can be determined.

Additionally, algorithms in standard applications are optimized for the average case (e.g. the quicksort algorithm). Since the average case is only of low relevance in hard real-time systems, algorithms on those dynamic data structures should have an optimal (read: minimal) worst-case execution time. Therefore, we require (1) a model that allows the determination of WCETs of the generated code and (2) we should generate code so that the WCETs are optimized.

Current WCET analysis techniques are restricted to imperative programming languages. Dynamic, object-oriented programming languages are not addressed at all. Buttazzo even demands to avoid dynamic data structures in real-time systems [5].

The standard approach in WCET analysis is to analyze the longest executable path, to map each instruction of this path to *elementary operations*, and to determine the WCETs of these elementary operations. The elementary operations can be for example assembler instructions as in the [6] or Java Byte Code instructions as in [1].

In [6], the WCET of a fragment of generated C code is determined by summing up the number of processor cycles each C instruction's corresponding assembler instruc-

†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.

tions require. For loops, the *worst-case number of iterations (WCNIs)* is derived from a statechart model to obtain the maximum number of executions of the loop-bodies.

[15] describes multiple existing approaches that use different annotations to specify the WCNIs and thus the longest executable path. All described approaches are restricted to imperative programming languages, that do not provide or use dynamic data structures. Further, the authors explain that an execution time analysis on the hardware level, which considers techniques like caching or pipelining, is required to avoid a too pessimistic estimation.

We present in this paper how code generation from UML class diagrams can be improved such that the resulting source code can be employed in hard real-time systems as representation for complex content. We propose to specify query and update tasks on this content as story patterns [7, 19]. Therefore, we present how source code with a minimal worst-case execution time can be synthesized from story patterns. We generate code for C++. We assume that worst-case execution times are known for all calls to external functions. In conformance to standard approaches for real-time systems, we assume that all memory is allocated at the start of execution.

In the next section, we present the example which is used in the remainder of the paper. Section 3 contains foundations which are required for predictable real-time behavior. Based on this foundations, we present in Section 4 how worst-case execution times are determined. In Section 5, the approach for computation of optimal worst-case execution times is presented. We conclude in Section 6 and present possible future work.

2. EXAMPLE

In the new transport system developed in Paderborn[1] autonomous vehicles drive on a railway system. Communication is required between the shuttles for coordination purposes, for example for building convoys to reduce the air resistance and thus the general power consumption.

The railway system is divided into multiple sections, each coordinated by a so-called *Registry*. Before entering a track section, a shuttle has to register at the corresponding registry. The registry collects information about the shuttle's current position and velocity and broadcasts this information to the other shuttles via wireless communication.

In this paper, we regard a situation as shown in Figure 1 when two shuttles move towards the same joining switch. The shuttles need to coordinate how to pass the switch in order to avoid a possible collision. Obviously, this has to be finished before they reach the switch. Thus, this coordination problem is subject to hard real-time requirements.

In order to recognize and to handle such situations, we use an ontology based topology for every shuttle to store environmental information in a discretized manner. On the one hand, each shuttle recognizes changes in its environment by sensors, on the other hand, it periodically receives updates of this environmental information from the registry as described in [10]. Figure 2 shows the UML class diagram of a shuttle's topology.

Every shuttle knows a registry that is liable for the set of tracks on which the shuttle is located. A CommunicationRule is like an instruction to handle a certain problem. Therefore,

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

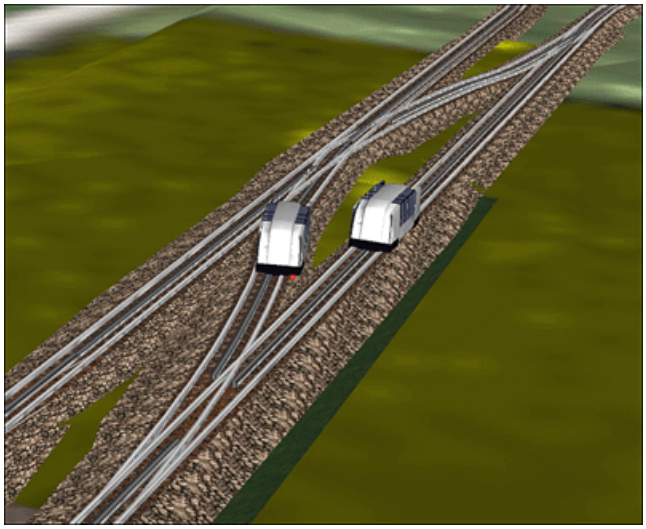

Figure 1: A possible collision of two shuttles at a switch

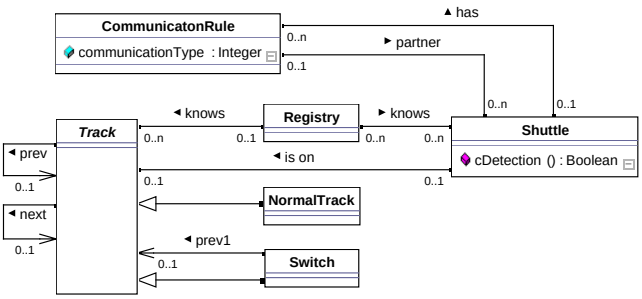

Figure 2: UML class diagram for the shuttle ontology

it has a partner association that describes which shuttles are involved in the respective problem. It also provides a type to classify the required coordination and thus the problem. Every shuttle provides a cDetection method that must be initiated if a shuttle is heading towards a switch. This method checks whether a situation might occur that causes a collision as illustrated in Figure 1. If a possible collision is detected, a CommunicationRule for avoiding the collision is created. This rule describes which shuttle has to slow down to avoid the collision. Of course, the shuttles may initiate a coordination, e.g. to buy or sell respectively the right of way. Figure 3 shows a story diagram that consists of one story pattern and specifies the method cDetection.

The behavior, specified by the story pattern of the story diagram, consists of two parts: First, an object matching searches for the situation that might cause a collision. This situation occurs when two shuttles are located on the tracks, that lead to a joining switch. If such an instance situation is matched, the second part of the behavior creates a CommunicationRule object with type = RIGHT_OF_WAY where RIGHT_OF_WAY is a constant for right of way. It also creates two links between the involved shuttles. When a matching is found and the CommunicationRule is created, the story

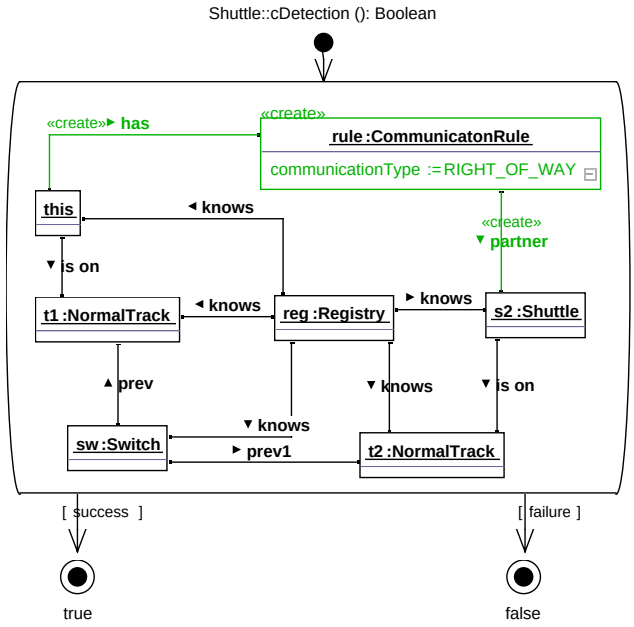

Figure 3: Story Diagram for collision recognition

diagram returns a true value, false otherwise. Both shuttles execute complementing story pattern which guarantee that one shuttle has right of way and the other one has to wait.

The mentioned coordination can be specified by another story diagram or by a Real-Time Statechart [4, 9, 2] that uses the return value of the cDetection method as transition trigger. Further aspects of real-time systems, like for example the communication, is out of the scope of story patterns and is handled for example in [11].

3. PREDICTABLE REAL-TIME BEHAVIOR

As indicated in the introduction, unbounded data structures lead to unpredictable real-time behavior. As class diagrams describe unbounded data structures and thus unbounded data structures are used to implement class diagrams, like the one from Figure 2, story patterns that operate on these data structures do not show predictable real-time behavior. Therefore, a WCET of a story pattern cannot be derived automatically from the model.

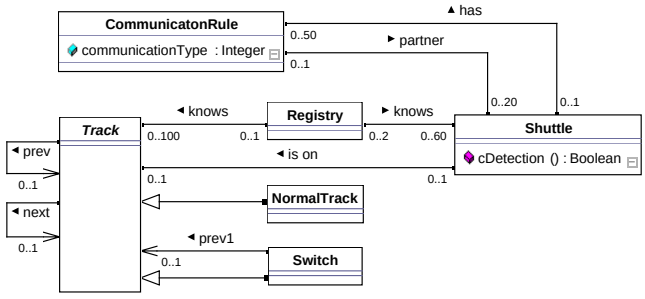

Figure 4: Class diagram with fixed maximum multiplicities

To overcome this limitation, we define so-called *fixed maximum multiplicities* in a class diagram, as shown in Figure 4. Note that the multiplicities, labeled with n in Figure 2, are replaced by concrete values in Figure 4. This enables an implementation using data structures with upper bounds. These upper bounds determine a *worst-case number of iterations (WCNIs)* when searching in these data structures which leads to predictable real-time behavior. This model-based development approach, combined with automatic code-generation leads to a well-structured implementation with analyzable nested loops and loops with fixed termination conditions.

Further, we make use of the factory pattern [8] to avoid dynamic resource allocation and deallocation after initialization time. As we know the implementation scheme of the access methods of the factory pattern and the implementation scheme of the code fragments that realize the story pattern, we derive their WCETs simply by adding the WCETs of the corresponding elementary operations.

There are several elementary operations on dynamic data structures in order to execute a story pattern. Elementary operations are creation and deletion of objects, adding and removing objects from different data structures, writing and reading attributes, and comparing objects. For each of these elementary operations, we use a runtime measurement tool executing a worst-case scenario running on the selected hardware platform. From this runtime measurement tool, we get the required WCETs. As different types of data structures are used (e.g. TreeSet, HashSet, LinkedList, ...), we compute the WCETs for the different data structures using different worst-case scenarios. The data structures used in the worst-case scenario have the maximum size as specified by the maximum multiplicities in the class diagram. As we know the code of the data structures, we also know the worst-case path when operating on them. In the future, the worst-case scenarios will be extended to capture degenerated data structures for a more precise WCET estimation.

The WCET of a story pattern does not only depend on the WCETs of its single code fragments and on the WCNIs when searching in data structures. The problem of WCET determination for story patterns is more complicated, because the order in which the elements of a story pattern are matched has significant impact on the resulting WCET as (partly) nested iterations can occur:

Multiple different matching sequences that lead to different WCETs exist because story patterns can contain bidirectional cycles. In the example story pattern of Figure 3, there exist several uni- and bidirectional cycles. For example, if the only bound object is this, this → t1 → reg → this is a bidirectional cycle because we also have the possibility to choose this → reg → t1 → this to match this part of the story pattern. For example, a unidirectional cycle is reg → sw → t1 → reg because the association between Switch and NormalTrack (which is a Track) is unidirectional.

The reason why different matching sequences usually lead to different WCETs is because different matching sequences can have different WCNIs. When, for example, a link between a Registry and a Shuttle instance is specified, starting at the Registry object and binding the Shuttle object requires a search in a data structure with 60 as upper bound. Binding the Registry object from the Shuttle object requires just a search in a data structure consisting maximal of 2 instances. In this case the algorithm, which determines the matching

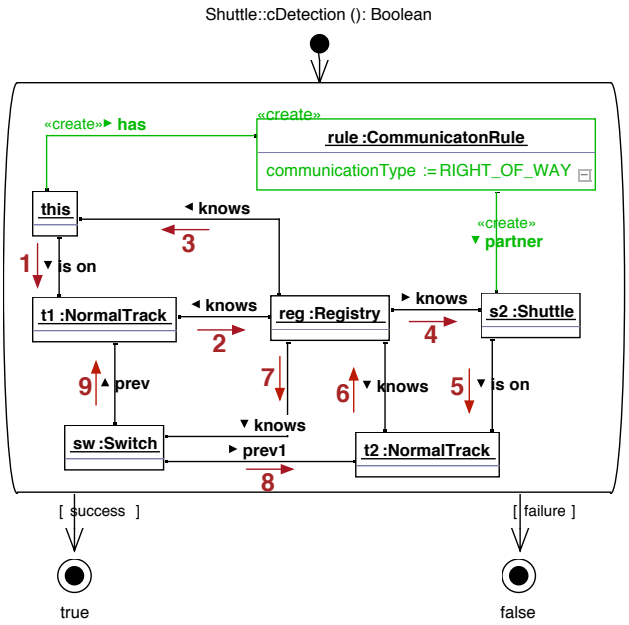

Figure 5: Story diagram and any matching sequence

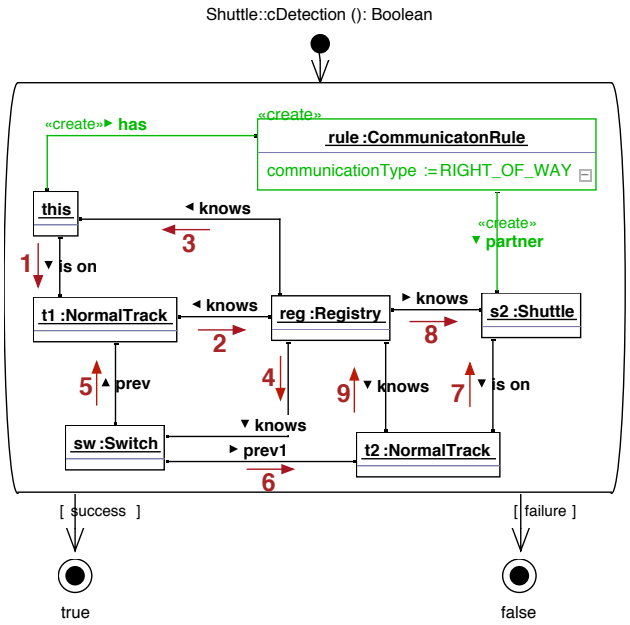

Figure 6: Story diagram with a better matching sequence (optimal)

sequence, has two possibilities that lead to the same instance matching but use different sequences.

Another reason why different matching sequences usually lead to different WCETs is that the matching process explores in the worst-case a path for each existing instance when binding an instance that is connected to a bound instance via a to-many association. To obtain an optimal WCET, the number of such paths has to be minimized. This is achieved by first respecting the path via associations with low multiplicities.

In Figures 5 and 6, the arrows with associated numbers represent different matching sequences for the shown story pattern. The two different strategies to perform the matching lead to different WCETs due to the different maximal sizes of the data structures as described two paragraphs before. As there exist multiple possible strategies to perform the matching, we should choose a strategy, that leads to an *optimal WCET*, i.e. a WCET that is as small as possible.

As we specified fixed maximum multiplicities and thus know the upper bounds of the corresponding data structures, we can determine a matching sequence so that the matching will use a minimum of comparisons when searching data structures and thus leads to the optimal WCET. In the next section is described how to determine the WCET for a matching sequence of a story diagram. Section 5 describes how to find the optimal matching sequence.

4. WCET DETERMINATION

In order to calculate and optimize the WCET of a story pattern, we introduce the so-called *story graph* [17]. This graph consists of different types of edges respecting that there are different kinds of checks to be performed during matching: For example, starting at the this object binding object t1 (step 1 in Figure 6) and then binding binding reg (step 2) is simple as the corresponding associations are to-one associations. For step 3, it is just a simple check for existence of a link is required, as the source and the target objects are already bound. Binding sw from reg in step 4 requires the search in a data structure, as it is not a to-one but a to-many association.

Before explaining further details like *story graph* creation, edge selection mechanism, timing constraints and WCET calculation with a *story graph*, a formal definition of the *story graph* is given in Definition 1.

Definition 1 *Let $G = (V, E)$ be a story graph with V the nodes, $E \subseteq (V \times V \times \mathbb{N}^3 \times \mathbb{N} \times E_s \times T)$ the edges and let $G_s = (V_s, E_s)$ be a story pattern. Each node $v_s \in V_s$ is mapped to a node $v \in V$ and each edge $e_s \in E_s$ is mapped to one or multiple edges $e \in E$ (see below). Thus, it holds $|V| = |V_s|$ and $|E| \geq |E_s|$. An edge $e = (s, t, w, c, L_e, t_e) \in E$ consists of the following elements: $s \in V$ is the source node of the edge e. $t \in V$ is the target node of the edge e. $w = A_{tt} = (w_f, w_d, c_a)$ is defined as AnalysedTypeTime which includes all timing information required to compute the WCET when choosing the edge e. $c \in \mathbb{N}$ is the maximum number of iterations that is required for binding the edge e. $L_e = \{e_{s1}, ..., e_{sn}\} \subseteq E_s$ is a set of Link/MultiLink-references of the story pattern G_s associated with the story graph edge e. $t_e \in T = \{$ BindNormal, BindOptional, CheckIsomorphism, CheckLink, CheckAttribute, CheckConstraint, CheckNegativeLink, CheckNegativeNode $\}$ is the type of the edge $e \in E$.*

V_s and E_s define the nodes and edges of a story pattern. See [19] for a detailed formalization of story patterns. An AnalysedTypeTime $w = A_{tt} = (w_f, w_d, c_a)$ contains runtime information. w_f is a fixed execution time that occurs due to a code fragment before starting a possible loop. w_d is the execution time for a single loop iteration. The number

of iterations is stored in c_a that results from the defined multiplicity of the related association in the related class diagram introduced in section 3. c_a should not be confound with c. In most cases they are equal, but there is an exception when these values differ. If there is an edge $e_i \in E$ with $t_i = \{\text{CheckNegativeNode}\}$ and the AnalysedTypeTime w_i then $c_i = 1$, but c_{ai} is the number of iterations that is derived from the defined exact multiplicity of the related association. The generated code that is necessary for checking negative nodes never starts a further nested loop, but for WCET computation of the code fragment for the negative node check, the fixed multiplicity is necessary and stored in c_{ai}. c is only used when calculating the WCET $K(L)$ of a matching sequence L (cf. Definition 2).

Every *story graph* edge $e \in E$ can optionally have one or more associated Link/MultiLink-references $L_e \subseteq E_s$. L_e does not influence WCET computation, but provides information required for implementation issues.

$t_e \in T$ describes the classification of a *story graph* edge $e \in E$. As defined in Definition 1 the set T includes eight classification types. These classification types describe groups of link types of a story pattern. For example, $t_e = \{\text{BindNormal}\}$ defines a link which describes a normal matching of an instance (except the links with optional condition). Every classification type describes indirectly a pre selection criterion and a post selection effect used while finding an optimal matching sequences described in the next section. The pre selection criterion describes which *story graph* edge $e \in E$ is available for selection. In every computation step, the algorithm for optimization has to choose a *story graph* edge $e \in E$ which was not considered before. The selection of a *story graph* edge $e \in E$ affects the *story graph* in a way that is implicitly encoded in the classification type $t_e \in T$ of the selected edge $e \in E$ (e.g. after binding a node via a link, this link does not need to be checked any more) what is called the post selection effect.

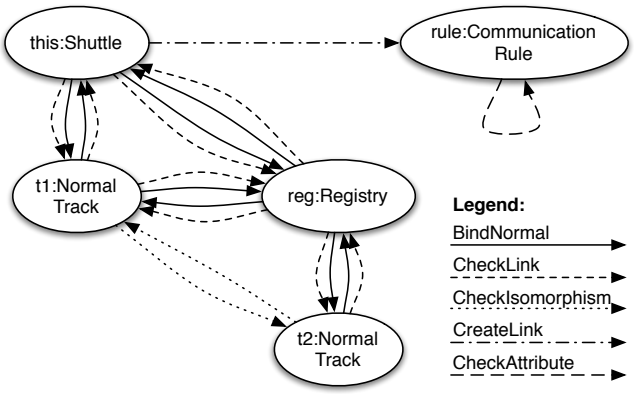

Figure 7: Cut-out of the resulting *story graph* from the story pattern example

The *story graph*, resulting from our example story pattern inside the story diagram introduced in Figure 3, consists of 7 nodes and 39 edges. Due to lack of space, we present just a cut-out of the *story graph* consisting of 5 nodes and 20 edges illustrated in Figure 7. This is adequate to explain the importance of the *story graph*.

Note that the *story graph* edges $e \in E$ are not inevitably related to story pattern edges $e_s \in E_s$. For example, there is no Link/MultiLink-reference $e_s \in E_s$ for attribute checks and assignments, but the *story graph* contains an edge $e_i \in E$ with $t_i = \{$ CheckAttribute $\}$ with the rule node as source and as target. For every story pattern Link/MultiLink-reference $e_s \in E_s$, several *story graph* edges $e \in E$ are created. For example, for every $e_i \in E$ and $t_i = \{$ BindNormal $\}$ a corresponding *story graph* edge $e_j \in E$ with $t_j = \{$ CheckLink $\}$ exists also. As mentionend above, either the BindNormal or the CheckLink link is taken for the matching sequence. The *story graph* edges $e_i \in E$ and $t_i = \{$ CheckIsomorphism $\}$ exists between all objects which have the same class diagram type.

The WCET for a story pattern and a specific matching sequence is determined as described in Definition 2.

Definition 2 *Let $K(L)$ be the WCET for a solution L of a story graph $G = (V, E)$ with $L = (e_1, ..., e_n)$, $e_i = (s_i, t_i, w_i, c_i, L_{ei}, t_{ei}) \in E$, $w_i = (w_{fi}, w_{di}, c_{ai})$ and $c_0 = 1$. Then $K(L) = \sum_{i=1}^{n} \left[(w_{fi} + w_{di} \cdot c_{ai}) \cdot \left(\prod_{j=0}^{i-1} c_j\right)\right] + pC(G, L)$.*

A solution L is an n-tuple of *story graph* edges $e \in E$. This n-tuple defines the matching sequence described in Section 3. As this matching sequence should be used for code generation of a story pattern, it can be translated into a regular matching sequence for the story pattern (the translation needs the Link/MultiLink-references stored in L_{ei} of every *story graph* edge $e_i \in E$). The function $pC(G, L)$ returns a runtime that is caused by code fragments that do not affect the runtime of the related matching sequence related to L. These code fragments are executed before or after a story pattern is matched successfully. For example, object and Link/MultiLink-reference deletion take place after successful matching. For $i = 1..n$ the execution time of the *story graph* edge e_i is multiplied with the number of how many times it will be checked in worst-case (WCNI).[2] This is described by the product inside $K(L)$ which is the number of iterations of the considered *story graph* edge e_i. By building the sum of these execution times and adding $pC(G, L)$, we get the WCET of the considered matching sequence L.

5. WCET OPTIMIZATION

At the end of Section 3, we stated that story patterns could have many valid matching sequences. This means that there exists at least one matching sequence in the set of all possible matching sequences that will take a minimum of runtime during its execution in the worst-case. So, for an optimal WCET, the determination of a solution L is required that minimizes $K(L)$.

In order to determine $min(K(L))$, we use a brute force back tracking search method, as listed in Figure 8. This algorithm determining the optimum requires exponential runtime in relation to the number of Link/MultiLink-references of a story pattern and the existing bidirectional cycles. It uses recursion to compute valid solutions L. A solution L is only valid when its WCET is less than the WCET of the best solution the algorithm found up to this point. Further, the solution has to contain all necessary *story graph* edges $e \in E$

[2]Due to a technical issue in the product function, we initially start with $c_0 = 1$.

```
1: s ← Defined WCET of the engineer
2: AL ← ApproximatedSolution(G)
3: k ← K(AL) ∨ defined upper bound of the engineer
4: minimum ← Ø
5: L ← Ø ∧ L.valid = true
6: function OptimalSolution(G, L)
7:     Sort all edges from i = 1...n ascending by w_fi + (w_di · c_ai)
8:     if (Not all edges e_i ∈ E in G marked) ∧ (Edges still reachable) ∧ (L.valid = true) then
9:         for All reachable edges e_i ∈ E in G do
10:            G' = (V', E') ← G = (V, E)
11:            L' ← L ∘ e_i
12:            Process all necessary markings e'_i ∈ E' of G'
13:            if K(L') < k then
14:                OptimalSolution(G', L')
15:            else
16:                L'.valid ← false
17:            end if
18:        end for
19:        if still edges e_i available then
20:            L.valid ← false
21:        end if
22:    end if
23:    if L.valid = true then
24:        minimum ← L
25:        k ← K(minimum)
26:    end if
27:    if k ≤ s then
28:        Terminate
29:    end if
30: end function
```

Figure 8: Algorithm for $min(K(L))$ determination

to become valid. Thus, after termination of the algorithm, the invariant $minimum = min(K(L))$ is true.

As a method with exponential time might lead to problems in practice, we improved the algorithm as shown in Figure 8: We use a heuristics (line 2), we ensure a monotonic decreasing of the upper WCET bound (line 3, 13, 25), and we introduce a lower WCET bound (line 1, 27).

First of all, a heuristics [17] is applied that leads to acceptable values for the WCET in the average case shown in Figure 9. The function *OptimalSolution* uses this function *ApproximatedSolution* to determine a solution L so that its WCET $= K(L)$ can be used as first upper bound. This cuts down the search space of possible solutions L at the beginning of OptimalSolution. OptimalSolution will recognize solutions as infeasible as soon as the execution time is greater or equal the heuristics WCET. This way of using a first upper bound and then decreasing the upper bound monotonic reduces the computation time significantly.

Usually, it is just required to obtain an implementation with a WCET that fits in a specific timing interval or just a given WCET from a requirement specification needs to be fulfilled. Thus, the engineer may define a target value for the WCET (s in Figure 8). The algorithm terminates when it determined a matching sequence that leads to a WCET that is below this target value. Obviously, this WCET can be larger than the optimal WCET.

```
1: function ApproximatedSolution(G)
2:     L ← Ø
3:     Sort all edges from i = 1...n ascending by w_fi + (w_di · c_ai)
4:     for Not all edges e_i in G are marked do
5:         for Unmarked edges reachable do
6:             Choose possible edge e_i from S(G) with smallest w_fi + (w_di · c_ai)
7:             L ← L ∘ e_i
8:         end for
9:     end for
10:    return(L)
11: end function
```

Figure 9: Algorithm to determine a first matching sequence

Tests showed that due to the improvements, a solution for a common story pattern can be found in acceptable time. Figure 10 illustrates the distribution of the different WCETs of the example shown in Figure 3. The figure shows the WCET values and the number of matching sequences with the respective WCET. The WCETs unit is milliseconds and is listed logarithmic.

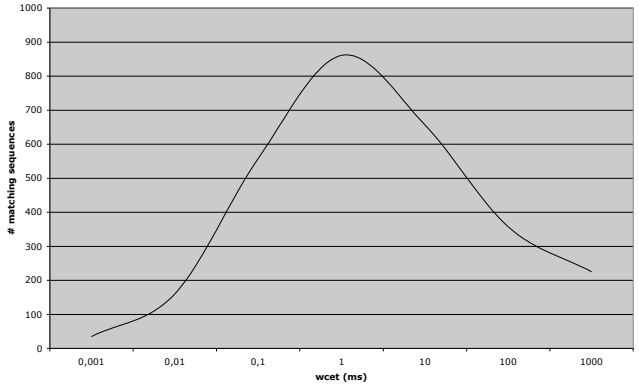

Figure 10: Frequency distribution chart of matching sequences and their WCETs

We see that most solutions have a WCET in the middle of minimum and maximum. Without the described improvements of the algorithm, the computation of the optimal matching sequence took about eight minutes on a 900 MHz PowerPC 750fx processor. Using the heuristics and monotonic decreasing of the upper bound it took about three seconds and only three possible matching sequences were found till optimum. Figure 11 shows the improvement of the example's WCET in relation to the time needed for optimization. Note that the x-axis is increasing exponentially.

The black line shows the temporal development of the best solution during the computation process. The grey line shows the optimal WCET that could be possible. This points out that using monotonic decrease of the upper bound is heavily decreasing the number of possible matching sequences.

For the evaluation shown in Figure 12, an abstract story pattern with twelve Link/MultiLink-references is used. The figure shows the improvement of the WCET in relation to

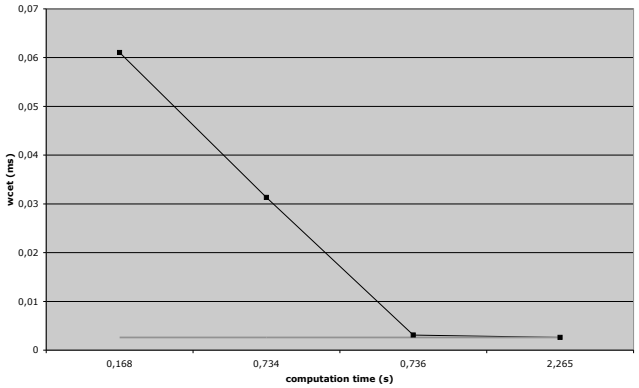

Figure 11: Improving the WCETs with the restricted algorithm

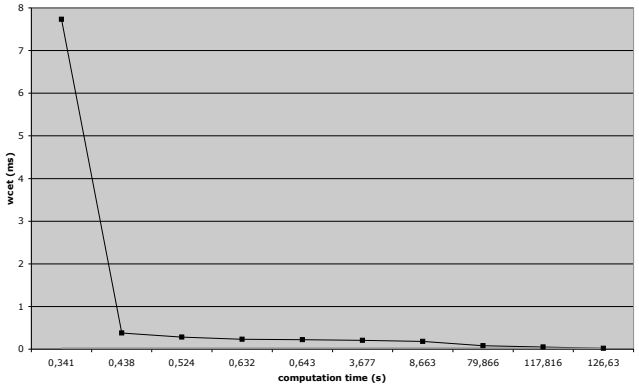

Figure 12: Improving the WCETs with the restricted OptimalSolution but with an abstract story pattern

the runtime of the optimization algorithm. Figure 13 shows a chart that results from computation time measurements of story diagrams with one abstract story pattern with a high number of bidirectional cycles. The x-axis describes the number of Link/MultiLink-references of the story pattern and the y-axis the OptimalSolution computation time in seconds. The diagrams show that story patterns with more then twelve Link/MultiLink-references require long computation times. Note that the computation time does not only depend on the number of Link/Multilink-references, but also on the number of bidirectional cycles which also increase the number of possible matching sequences.

6. CONCLUSIONS & FUTURE WORK

Graph like structures are required for storing context and local knowledge in future complex intelligent and adaptive technical systems. Story patterns are an appropriate modeling language for modifying graph like structures. In order to satisfy safety and hard real-time requirements, worst-case execution times for the execution of story patterns are required. We presented in this paper an approach which (1) determines these worst-case execution times on a given hardware and (2) computes an optimal worst-case execution time based on an optimal search order of the story pattern elements. The computation of the optimal order consists of two steps. In the first step, a heuristics is used in order to find an optimal search order for the average case. Thereafter, in step two, a brute force algorithm is employed to find a better solution than provided by the heuristics.

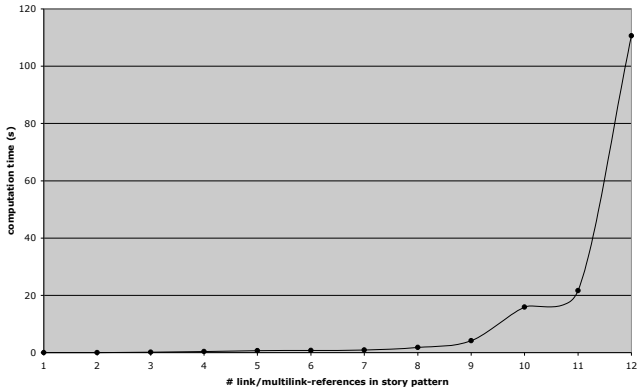

Figure 13: Computation times in relation to the number of Link/MultiLink-references of a abstract story pattern

To improve our WCET optimization algorithm, we plan to respect knowledge about the minimal remaining costs in the algorithm in a branch-and-bound manner. Further, we plan to support the WCET determination of story diagrams, consisting of multiple story patterns as well. Therefore, we plan to integrate our approach with the MAXT approach [16] that requires the specification of the WCNIs for every activity in the story diagram and their WCETs. We will use our algorithm for single story patterns to determine the single activities' WCETs. In [16], the WCET for the cyclic flow graph is then computed with integer linear programming (ILP).

Story charts [12] are an extension of standard UML state machines by story pattern. The states are enriched by story patterns as do methods. Story charts lack appropriate notions for time. Real-Time Statecharts [4, 9, 2] are an appropriate state based modeling notation for the specification of real-time behavior. The presented WCET determination and optimization approach will be used in order to integrate story patterns into Real-Time Statecharts. In addition to the usage of story patterns in story charts, we will not only support story patterns as behavior specification for do methods, but for all kind of actions (entry and exit methods as well as transition actions).

REFERENCES

[1] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS 2000)*, 2000.

[2] S. Burmester and H. Giese. The Fujaba Real-Time Statechart PlugIn. In H. Giese and A. Zündorf, editors, *Proc. of the first International Fujaba Days 2003, Kassel, Germany*, volume tr-ri-04-247 of *Technical Report*, pages 1–8. University of Paderborn, 2003.

[3] S. Burmester, H. Giese, and O. Oberschelp. Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In *Informatics in Control, Automation and Robotics*. Kluwer Academic Publishers, 2005. to appear.

[4] S. Burmester, H. Giese, and W. Schäfer. Model-driven architecture for hard real-time systems: From platform independent models to code. In *Proc. of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany*, pages 1–15, November 2005.

[5] G. C. Buttazzo. *Hard Real Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer international series in engineering and computer science : Real-time systems. 1997.

[6] E. Erpenbach. *Compilation, Worst-Case Execution Times and Scheduability Analysis of Statechart Models*. Ph.D.-thesis, University of Paderborn, Department of Mathematics and Computer Science, 2000.

[7] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. LNCS 1764, pages 296–309, November 1998.

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

[9] H. Giese and S. Burmester. Real-Time Statechart Semantics. Technical report, 2003.

[10] H. Giese, S. Burmester, F. Klein, D. Schilling, and M. Tichy. Multi-Agent System Design for Safety-Critical Self-Optimizing Mechatronic Systems with UML. In B. Henderson-Sellers and J. Debenham, editors, *OOPSLA 2003 - Second International Workshop on Agent-Oriented Methodologies*, pages 21–32, Anaheim, CA, USA, Center for Object Technology Applications and Research (COTAR), University of Technology, Sydney, Australia, Oct. 2003.

[11] 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.

[12] H. J. Köhler, U. A. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. pages 241–251. ACM Press, 2000.

[13] D. J. Musliner, R. P. Goldman, M. J. Pelican, and K. D. Krebsbach. Self-Adaptive Software for Hard Real-Time Environments. *IEEE Inteligent Systems*, 14(4), July/Aug. 1999.

[14] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999.

[15] P. P. Puschner and A. Burns. Guest Editorial: A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2/3):115–128, May 2000.

[16] P. P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times - A Graph-Based Approach. In *Real Time Systems, 13*, Technical Report, pages 67–91. Springer Link, Kluwer Academic Publishers, July 1997.

[17] A. Seibel. Story Diagramme für Eingebettete Echtzeitsysteme. Bachelor Thesis at University of Paderborn, Department of Computer Science, Paderborn, Germany, February 2005.

[18] J. Sztipanovits, G. Karsai, and T. Bapty. Self-adaptive software for signal processing. *Commun. ACM*, 41(5):66–73, 1998.

[19] A. Zündorf. Rigorous Object Oriented Software Development. Habilitation Thesis at University of Paderborn, Department of Computer Science, Paderborn, Germany, 2001.