Fujaba Days 2009

Proceedings of

the 7th International Fujaba Days

Eindhoven University of Technology, The Netherlands
November 16-17, 2009

Editor: Pieter Van Gorp

Technische Universiteit
Eindhoven
University of Technology

1 Introduction

Fujaba is an Open Source UML CASE tool project started at the software engineering group of Paderborn University in 1997. In 2002 Fujaba has been redesigned and became the Fujaba Tool Suite with a plug-in architecture allowing developers to add functionality easily while retaining full control over their contributions.

Multiple Application Domains 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, and Darmstadt for supporting (1) reengineering, (2) embedded real-time systems, (3) education, (4) specification of distributed control systems, (5) integration with the ECLIPSE platform, and (6) MOF-based integration of system (re-) engineering tools.

International Community According to our knowledge, 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 requests for more functionality and extensions.

Therefore, the 7th International Fujaba Days aimed 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.

1.1 Organizing Committee

- Organization: Pieter Van Gorp, Eindhoven University of Technology (TU/e)
- Support: Jochem Vonk (Technical Administration) and Annemarie van der Aa (Secretary)

1.2 Program Committee

- Jendrik Johannes (Technische Universität Dresden, Germany)
- Jürgen Börstler (University of Umea, Sweden)
- Holger Giese (Hasso-Plattner-Institut Potsdam, Germany)
- Pieter van Gorp (University of Antwerp, Belgium)
- Jens Jahnke (University of Victoria, Canada)
- Mark Minas (University of the Federal Armed Forces, Germany)
- Manfred Nagl (RWTH Aachen, Germany)
- Arend Rensink (University of Twente, Netherlands)
- Andy Schürr (TU Darmstadt, Germany)
- Wilhelm Schäfer (University of Paderborn, Germany)
- Bernhard Westfechtel (University of Bayreuth, Germany)
- Albert Zündorf (University of Kassel, Germany)

2 Program and Table of Contents

Repository Integration / Middleware (day 1)

1. Research Papers:
    - Jendrik Johannes. *Developing a Model Composition Framework with Fujaba An Experience Report* p. 1
    - Holger Giese, Stephan Hildebrandt and Andreas Seibel. *Feature Report: Modeling and Interpreting EMF-based Story Diagrams* p. 5
2. Tool Demonstration:
    - Nina Geiger, Christoph Eickhoff, Marcel Hahn, Ingo Witzky and Albert Zündorf. *Future Web Application Development with Fujaba* p. 51

Pattern Matching and Rewriting (day 1)

1. Research Papers:
    - Mirko Seifert and Christian Werner. *Specification of Triple Graph Grammar Rules using Textual Concrete Syntax* p. 10
    - Markus von Detten and Marie Christin Platenius. *Improving Dynamic Design Pattern Detection in Reclipse with Set Objects* p. 15
2. Tool Demonstration
    - Jörn Dreyer and Albert Zündorf. *NT2OD - From natural text to object diagram* p. 56

Modeling Languages (day 1)

1. Research Papers:
    - Thomas Buchmann and Alexander Dotor. *Mapping Features to Domain Models in Fujaba* p. 20
    - Ruben Jubeh and Albert Zündorf. *PropertyChange Events meet Fujaba Statecharts* p. 25
    - Bart Meyers and Hans Vangheluwe. *Evolution of Modelling Languages* p. 29
2. Tool Demonstration
    - Alexander Dotor, Thomas Buchmann and Martin Klinke. *Supporting Modeling in the Large in Fujaba* p. 59

Real-Time (day 2)

1. Research Papers:
    - Pieter Van Gorp, Ruben Jubeh, Bernhard Grusie and Anne Keller. *Fujaba hits the Wall(-e)* p. 34
    - Tobias Eckardt and Stefan Henkler. *Synthesis of Component Behavior* p. 40
    - Claudia Priesterjahn and Matthias Tichy. *Modeling Safe Reconfiguration with the Fujaba Real-Time Tool Suite* p. 45
2. Tool Demonstrations:
    - Stefan Henkler, Moritz Breit, Christopher Brink, Markus Bger, Christian Brenner, Kathrin Brker, Uwe Pohlmann, Manuel Richtermeier, Julian Suck, Oleg Travkin and Claudia Priesterjahn. *$FRiTS^{Cab}$: Fujaba Re-Engineering Tool Suite for Mechatronic Systems* p. 64
    - Chrisitan Heinzemann, Stefan Henkler and Albert Zündorf. *Specification and Refinement Checking of Dynamic Systems* p. 69

3 Research Papers

This section contains all the research papers that were accepted for presentation by the program committee. The acceptance ratio for Fujaba Days 2009 is 2/3.

Developing a Model Composition Framework with Fujaba – An Experience Report

Jendrik Johannes*
Technische Universität Dresden
Software Technology Group
01062 Dresden, Germany
jendrik.johannes@tu-dresden.de

ABSTRACT

Reuseware is an open-source model composition framework for composing models defined in arbitrary Ecore-based languages. In its four years of development, Reuseware has experienced many extensions and refactorings due to the integration of new research results and requirements. One year ago, a redevelopment of Reuseware's core was started. Thanks to its EMF code generation, Fujaba was introduced as a new development tool into Reuseware's development toolchain to replace major parts of Java coding through story driven modelling. With this we solved problems with behavior modelling and code generation we faced in the development so far. This paper summarizes our experiences in developing with Fujaba and suggests improvements for Fujaba and its EMF code generation based on that.

1. INTRODUCTION

The Reuseware project[1] was started at the Software Technology Group of TU Dresden in 2005 as successor of the COMPOsition SysTem (COMPOST) framework[2]. COMPOST implemented the concepts of Invasive Software Composition (ISC) [1] for Java and XML. ISC is a static software composition approach that can act as a basis to implement a variety of composition techniques for arbitrary languages. While COMPOST showed the applicability of the approach for Java and later for XML, it was completely hand-written. Adapting it for XML for instance, took considerable effort.

The aim of the Reuseware project is to build a framework for ISC where new languages can be plugged in without manual coding only by providing a grammar or a metamodel of the language. It was clear from the beginning that Reuseware should be developed as an Eclipse extension to profit from the features already provided in the open IDE. The second thing required was a meta language to describe languages that can be plugged in. After several experiments, Ecore—an implementation of the OMG's EMOF standard [8]—of the Eclipse Modelling Framework (EMF) [10], was chosen. We decided for Ecore because of its standard conformance and code generation facilities that integrate nicely into Eclipse. Furthermore, since we initially focused on textual languages, we needed some grammar processing tooling on top of Ecore. Thus, we developed EMFText [5] that was initially part of Reuseware.

*This research has been co-funded by the European Commission in the 6th Framework Programme project MODELPLEX contract no. 034081 (www.modelplex.org).

Eventually, Reuseware itself was developed using Ecore for its metamodels, EMFText for textual and GMF [3] for graphical specification languages. What proved to be most problematic was the metamodelling in Ecore. Since it does not support behavior modelling, we had to add operation bodies manually to the generated code. As a consequence of that, we ended up with muddled generated and hand-written code and an unnatural separation of methods into utility classes. After several iterations and experiments, it became clear in the end of 2008 that Reuseware needed a major redesign.

At that point, we saw the main problems of the implementations in 1) the too tight integration of generated and hand-written code 2) the implementation of model (i.e., graph) transformations in Java, which was unnatural, buggy and hard to maintain. For both issues we desired a generative solution. Fujaba with its story diagram paradigm—to model graph transformations—and its EMF code generation [2]—to generate operation bodies of Ecore models from story diagrams—was the ideal candidate for that. This paper summarizes our experience in redeveloping huge parts of Reuseware with Fujaba. Furthermore, it explains extensions we made to Fujaba's EMF code generation.

2. DEVELOPING REUSEWARE WITH FUJABA

Describing Reuseware in detail is out of scope of this paper (please refer to [4, 6, 7] and the Reuseware website[1]). Nevertheless, we present the architecture of Reuseware in Sect. 2.1 to clarify which modelling technologies are used and where Fujaba fits in. We explain how Fujaba was integrated into our development toolchain and how it was customized for our purposes in Sect. 2.2. Our experiences in modelling with Fujaba are then described in Sections 2.3–2.5.

2.1 Reuseware Architecture

In Reuseware, we distinguish two user roles: *composition system developers* and *composition system users*. A composition system implements a certain component and composition methodology. Module systems or aspect systems are examples of composition systems. Composition systems can usually only handle components (e.g., modules or aspects) written in a specific programming or modelling language. Integrating new languages or new types of components usually takes considerable effort. Reuseware however,

[1]http://www.reuseware.org
[2]http://www.the-compost-system.org

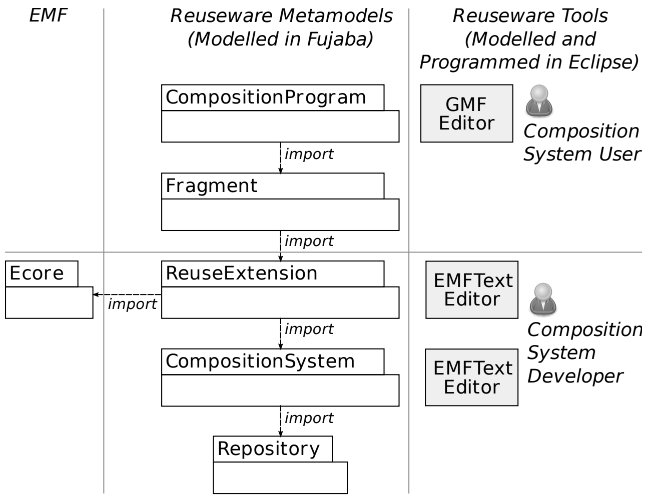


Figure 1: Reuseware architecture overview

is a framework which can be easily instantiated by *composition system developers* to support new composition systems for existing or newly developed languages. The such instantiated framework may then be used by *composition system users* to define components and composition programs (i.e., instantiations and compositions of components).

Reuseware is built around five metamodels shown in Fig. 1 (middle). Each metamodel is modelled as a class diagram in Fujaba. Each of these class diagrams is then translated to an Ecore model by Fujaba's EMF code generation. Instances of the metamodel are either derived by Reuseware or have a concrete syntax that can be used by a composition system developer or a composition system user directly.

The core metamodel is the *Repository* that models a package structure into which different types of elements can be placed. The repository metamodel is instantiated by the running Reuseware based on actual files in the workspace. It provides a component oriented viewpoint on these files for composition system users and developers alike.

A composition system developer may utilise two dedicated languages for composition system development—one language to define the concepts of a *CompositionSystem* and one language to specify where these concepts are found in a language defined as an Ecore metamodel. We call such a specification a *ReuseExtension* for a given language. Both, *CompositionSystem* and *ReuseExtension* language, have a textual syntax defined with EMFText. The *ReuseExtension* language embeds OCL [9] as expression language. These two metamodels are only used for specification and do not have operations that modify models. They therefore make use of story diagrams only to simplify access to elements (e.g., to find one item with a given name in a list).

A composition system user works with two kinds of artifacts: *Fragments* (the components in Reuseware) and *CompositionPrograms* (specifications for compositions of fragments). A fragment has a composition interface through which model elements are accessed (or modified) during a composition. Concepts for composition interfaces are modelled in the *Fragment* metamodel. Instances of that metamodel are created by interpreting *ReuseExtension* specifications on arbitrary EMF models. For this, story diagrams are used in combination with the OCL expressions embedded as strings in *ReuseExtension* models.

Composition programs can again be created by interpreting *ReuseExtension* models but also manually by a composition system user. It depends on whether the composition system developer specified a dedicated composition language for the composition system or decided to use the generic composition language of Reuseware. This composition language is defined in the *CompositionProgram* metamodel and has a graphical syntax defined with GMF. In any case, parts of a composition program can be derived and updated automatically. This is specified with story diagrams.

2.2 Setup and Customization

The development was performed with the SVN versions of Fujaba and the CodeGen2 plugin of Sept. 22nd 2008. To allow rapid development and testing, an ANT build script was written that deletes the previous generated code, distributes Ecore models and Java code generated by Fujaba correctly over several Eclipse plugin projects and triggers EMF's own code generation. Consequently, two clicks are needed to generate the code: Code generation in Fujaba and running the ANT script in Eclipse. The metamodelling was from now on performed in Fujaba. We continued to use Eclipse for EMFText modelling, GMF modelling and Java coding.

Some modifications of the code generation templates of Fujaba were needed to add features not yet supported by Fujaba's EMF code generation. In the following we summarize these modifications.

1. *Splitting the metamodel* The layered metamodel architecture depicted in Fig. 1 is realized by providing one Ecore file and one Eclipse plugin per metamodel. The dependencies between the plugins correspond to the dependencies between the metamodels. To support this splitting while preserving references between the metamodels, the code generation was adjusted.
2. *Referencing existing Ecore models* As one can see in Fig. 1 (left), some of the Reuseware metamodels depend on the Ecore metamodel (i.e., the *Ecore.ecore* file found in EMF) that is modelled in Ecore itself. We made this metamodel available inside Fujaba by importing the *org.eclipse.emf.ecore.jar* that contains the code generated from *Ecore.ecore*. This allows us at least to reference classes from that model in class diagrams. The templates, however, did not support *referenced* classes at all. We modified the templates to create references to *Ecore.ecore*, which exists in every EMF installation, when appropriate.
3. *List return types* In Ecore, return types of operations can be multiple (upper bound > 1). This is not supported by Fujaba and its EMF code generation. Since we needed this feature, we integrated it by introducing the stereotype *multiple* in Fujaba. The extended code generation sets the upper bound of an operation with

that stereotype to *. EMF expects the code of such operations to return an *org.eclipse.emf.util.EList< T>* where T is bound to the return type of the operation. Consequently, the developer of the story pattern of a *multiple* operation has to instantiate, fill and return such an *EList* manually using Java statements.

4. *String arrays* In the case of strings (and other primitive types) the above situation is better, because Fujaba offers explicit primitive array types (e.g., *StringArray*) in its standard library. We extended the code generation to translate Fujaba's *StringArray* into *EList<String>*. Similar conversion could be done for the other primitive types but are not needed in our case.

5. *Navigating operations as links* In story diagrams, we sometimes needed to navigate a virtual path rather than a direct reference in a model. Fujaba supports this through *path expressions*. The drawback of these expressions is that they are translated into an interpreter call which is only checked at runtime and always requires a path expression interpreter. We needed path expressions not for complex expressions, but to use the result of an operation as path or to access references of the Ecore metamodel, which are not known by Fujaba (because *Ecore.ecore* was imported as jar file as described above). This limited use of path expressions enabled us to modify the templates to translate a path expression into an operation call instead of calling the expression interpreter. Additional code ensures that the result of the operation is wrapped into an iterator as expected by the rest of the template.

6. *Support for eKeys* To improve the serialization of references between elements in XMI files, Ecore offers the *eKeys* concept. An *eKey* is essentially a primary key that identifies a model element (e.g., a name attribute is a good candidate for an *eKey*). The serialization then uses the *eKey* to identify cross-referenced elements—instead of using the positions of the element which is the default. Using positions can lead to problems when two XMI files reference each other and one is changed independent of the other. This is the case for the composition program GMF editor, where the layout information (e.g., position of boxes) is saved in a different file than the model elements. To support *eKeys*, we introduced the stereotype *ID* in Fujaba. If an attribute is stereotyped with *ID*, the extended templates define an *eKey* based on that attribute.

7. *Generate code with bound list parameters* A not vital but nice extension is the binding of type parameters in lists and iterators. Since Reuseware is developed in Java5, the generated code produced compiler warnings concerning unbound type parameters. When we cleaned up the code, we addressed all compiler warnings. One of this was to generate the type parameter binding, which was not difficult because all required type information is available during code generation.

To summarize, modifications 1 and 5 seem to be very specific for the development of Reuseware. All other extensions however, could be beneficial for other projects as well. It should be investigated, if and how these modifications can be integrated into the current Fujaba trunk templates.

2.3 General Development Experience

Despite the use of two different development environments, the development felt quite integrated. With the help of the above mentioned build script, changes in the Fujaba models are quickly updated in the Eclipse workspace. The overall generation process takes less than 10sec (on 2.33 GHz Intel Core 2 Duo) and requires only two clicks which is acceptable.

The adjustment of the templates themselves was manageable. It was done in a normal text editor. We have to admit that we did not bother to acquire a proper velocity template editor which could have eased the template modification. Very positive was however that the CodeGen2 templates could be updated in a running Fujaba, which made the debugging of changed templates easy.

Template customization was mainly a concern in the first development phase. Here frequent updates had to be done to support the desired behavior. Such adjustments however, became less frequent and no adjustment were required in the last six month—despite of ongoing development.

2.4 Working with Class Diagrams

Before we used Fujaba, the metamodels of Fig. 1 where developed directly in Eclipse. For this an open-source Ecore diagram editor provided by the TOPCASED project[3] was utilised. We compare our experiences using this editor with using the Fujaba class diagram editor.

The user experience of the Ecore diagram editor was in general not very good. 1) Often the editor feels unstable and sometimes a diagram looks different after we saved and opened it again. Most annoying was the weak support for automatic layouting such that drawing a straight line always was a challenging task. 2) Copy and paste was not supported very well. To perform such task, we used the graphical editor in combination with the tree Ecore editor that comes with Ecore itself (and supports copy and paste very well). However, keeping the such modified model synchronised with the diagram representation was also not a strength of the diagram editor. At some point, the diagram file could not be opened anymore and the whole layout was lost. 3) Another difficulty was the specification of bi-directional associations, which are modelled as two uni-directional references that are connected via an *opposite* relationship in Ecore. The editor did not provide a facility to specify or represent those reference together. Thus, specifying a bi-directional association was cumbersome and ugly in the diagram—it was represented by two lines.

Although the class diagram editing was not the reason to switch to Fujaba, we were very pleased that the class diagram editor of Fujaba overcame the weaknesses of the Ecore diagram editor. 1) Using the editor feels very smooth and stable. Lines between classes are drawn straight automatically when possible. 2) Copy and paste is supported to a high degree and we were always able to perform a desired restructuring without having to re-model anything. 3) Bi-directional associations can be defined in Fujaba naturally. The EMF code generation translates these associations into two references in the Ecore model just as we expected.

[3]http://www.topcased.org

2.5 Working with Story Diagrams

As mentioned, the main motivation to use Fujaba was to define metamodel operations as story diagrams. As in class diagrams, the combination of manual and automatic layout gives the user a smooth editing experience and even the (re)structuring of large diagrams was easy. Although we tried to model as much as possible, the openness toward calling Java code directly was vital to continue the work at places were it was not obvious how to model the functionality best. It was necessary to achieve the integration with existing code—mainly with the EMF and an OCL interpreter (cf. next section). The cut and paste functionality was also very useful in particular to refactor story diagrams.

We can make the following improvement suggestions for story diagrams based on our experience.

1. *Calling story diagrams* To call one story diagram from another one is currently only possible by calling the Java method that is generated from that story diagram. This feels unnatural, since one calls the generated code (i.e., the Java method) and not the story diagram (i.e., the UML operation) on the modelling level. Because we wanted to stay on the modelling level, we avoided to split story diagrams in the beginning and the diagrams grew unnecessarily large. Having an explicit mechanism to call story diagrams would improve the modelling experience here.

2. *List return types* Fujaba does not support a mechanism to declare a return type of an operation as *multiple*. One can set the return type to *FHashSet*, but this does not say anything about the type of the values that may be contained in the set and thus can not be properly processed by the EMF code generation. It is fine to work with the *multiple* stereotype extension we presented in Sect. 2.2, but having the capability directly integrated into Fujaba would be even nicer.

3. *Import of existing Ecore models* The Reuseware metamodels depend on the Ecore metamodel (cf. Fig. 1, left). The metamodel is modelled in Ecore itself (in an *ecore* file). As mentioned, we imported this metamodel by importing the jar file that contains the code generated from the metamodel. This gave us access to the metamodel types which was sufficient to model class diagrams. In story diagrams however, we could not model edges between instances of classes from the Ecore metamodel since associations were not extracted from the jar file. This information is however contained in the *ecore* file. Providing an import for *ecore* files into Fujaba (the inverse of the EMF code generation) would greatly enhance the integration of Fujaba and EMF and would allow people to define story diagrams for their existing Ecore models.

Recently, we refactored our story diagrams and split them into smaller diagrams such that each diagram fills one A4 page at maximum when printed. All Reuseware metamodels together now contain 61 story diagrams (and 73 classes).

[4]`http://www.eclipse.org/modeling/mdt/?project=ocl`

2.6 Tool Interoperability

Through Fujaba's EMF code generation, the tool integration worked very smooth. The Ecore models produced by the code generation could be handled by any other EMF based tool. As illustrated in Fig. 1, we used EMFText and GMF to build editors for three of our five metamodels.

In one metamodel (*ReuseExtension*) we allow the specification of OCL expressions as strings. To interpret these expressions, we use the MDT OCL interpreter[4], which works with Ecore models. We implemented a small *Evaluator* utility class with static methods that initialize the OCL environment and use it to evaluate the embedded expressions. We then use Fujaba's ability to refer to arbitrary Java classes and methods inside of story diagrams to call methods on the *Evaluator*. In particular, it is used in to evaluate *boolean guard expressions* of transitions and to derive values for *attribute assignments* of an object in a story activity.

3. CONCLUSION

In this paper we reported on the development of Reuseware with Fujaba. We conclude that using Fujaba significantly improved the development experience and the quality of the developed tooling. Thus using Fujaba was the correct decision. With story driven modelling we were able to increase the amount of modelling and to improve separation of generated and hand-written code. We hope that the extensions of the EMF code generation discussed in Sect. 2.2 and the story diagram improvements suggested in Sect. 2.5 can help to improve Fujaba in the future. This paper showed that Fujaba's EMF code generation is usable in practice to develop EMF-based tools with Fujaba and to profit from story driven modelling in EMF.

4. REFERENCES

[1] U. Aßmann. *Invasive Software Composition.* Springer, Secaucus, NJ, USA, 2003.

[2] L. Geige, T. Buchmann, and A. Dotor. EMF Code Generation with Fujaba. In *Proc. of the 5th International Fujaba Days.* University of Kassel, 2007.

[3] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Pearson Education, 2009.

[4] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On Language-Independent Model Modularisation. In *Transactions on Aspect-Oriented Software Development VI*, volume 5560 of *LNCS.* Springer, 2009.

[5] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and Refinement of Textual Syntax for Models. In *Proc. of ECMDA-FA '09*, volume 5562 of *LNCS.* Springer, 2009.

[6] J. Johannes. Controlling Model-Driven Software Development through Composition Systems. In *Proc. of NW-MODE '09.* Tampereen teknillinen yliopisto, 2009.

[7] J. Johannes, S. Zschaler, M. A. Fernández, A. Castillo, D. S. Kolovos, and R. F. Paige. Abstracting Complex Languages through Transformation and Composition. In *Proc. of MoDELS'09*, volume 5795 of *LNCS.* Springer, 2009.

[8] Object Management Group. MOF 2.0 core specification. OMG Document, Jan. 2006. `www.omg.org/spec/MOF/2.0`.

[9] Object Management Group. Object Constraint Language, Version 2.0, May 2006. `www.omg.org/spec/OCL/2.0`.

[10] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *Eclipse Modeling Framework, 2nd Edition.* Pearson Education, 2008.

Feature Report: Modeling and Interpreting EMF-based Story Diagrams

Holger Giese
Hasso-Plattner-Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482, Potsdam, Germany
holger.giese@hpi.uni-potsdam.de

Stephan Hildebrandt
Hasso-Plattner-Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482, Potsdam, Germany
stephan.hildebrandt@hpi.uni-potsdam.de

Andreas Seibel
Hasso-Plattner-Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482, Potsdam, Germany
andreas.seibel@hpi.uni-potsdam.de

ABSTRACT
In this paper, we report on the current state of development of our Eclipse Story-Driven Modeling tools. These tools are a graphical editor and an interpreter for Story Diagrams. The graphical editor provides useful features like model validation for Story Diagrams, and advanced editing features like syntax highlighting and code completion for OCL expressions. The interpreter was initially presented in [5]. There, we showed that the interpreter enables new areas of application for Story-Driven Modeling, improves the flexibility of applying Story Diagrams, but can also improve the performance of executing Story Diagrams. In the meantime, the interpreter evolved. Beside its basic features, a dynamic pattern matching strategy and compatibility to dynamic EMF, we introduced new features like support for map-typed references, containment links and further optimizations of Story Patterns at runtime.

Categories and Subject Descriptors
D.2.m [**Software Engineering**]: Miscellaneous; H.1.m [**Models and Principles**]: Miscellaneous

Keywords
Story Diagram, Interpreter

1. INTRODUCTION
Story Diagrams were introduced in [2]. They combine UML Activity Diagrams with graph transformation rules to graphically describe the search for patterns in an object graph and the creation and deletion of objects and links. They were initially implemented in the Fujaba CASE tool [8], which can be used to model Story Diagrams and generate executable code from Story Diagrams.

In the past years, the Eclipse Modeling Framework (EMF)[1] has become the industry standard in the area of model-driven software engineering based on the Eclipse platform. There are many tools supporting model-driven software development (e.g., openArchitectureWare[2]), model transformations (e.g., ATL[3]) or other activities regarding models that are based on EMF. This has lead to the situation that we need to support EMF in our current research projects in order to be compatible to existing tools. However, we still needed Fujaba to model Story Diagrams and generate executable code for further application. Although there are attempts to minimize this technology gap, e.g., an EMF-compatible code generator [3] or an EMF adapter for Fujaba models [7], it still poses an obstacle as we already outlined in [1] in more detail.

Another central drawback of Story Diagrams in Fujaba is the static pattern matching strategy, which can be a serious performance bottleneck. This strategy is determined at generation time based solely on the information available in the meta models only. The strategy prefers to-one links over to-many links to match objects. However, it does not distinguish between to-many links concerning their exact size. This information is only available at runtime. Therefore, the static pattern matching strategy is the same for all possible instance models.

To overcome these obstacles, we developed an interpreter for Story Diagrams [5] based on Eclipse and EMF.[4] Its main improvement is a dynamic pattern matching strategy. This dynamic pattern matching strategy leverages information available in the instance models and distinguishes between to-many links depending on their exact size. The performance can vary tremendously between dynamic and static pattern matching strategies. In [5] the results of a benchmark are presented, which showed the effect of different pattern matching strategies concerning the performance of the pattern matching process. The benchmarks showed that

[1]http://www.eclipse.org/modeling/emf/
[2]http://www.openarchitectureware.org
[3]http://www.eclipse.org/m2m/atl/
[4]The interpreter and graphical editor for Story Diagrams can be downloaded from our update site: http://www.hpi.uni-potsdam.de/giese/gforge/sam-update-site/site.xml. Prerequisites are Eclipse 3.5, EMF 2.5, GMF 2.2, Xtend 0.7 and OCL 1.3.

Fujaba-generated Story Diagrams with non-optimal pattern matching strategies can be much slower than the interpreter with dynamic pattern matching strategy.

Another promising feature of the interpreter is the compatibility to dynamic EMF. Dynamic EMF omits generated code. Instead, dynamic EMF objects are instances of a common class, *DynamicEObjectImpl*. These instances can be populated with data from, e.g., an XMI file or a database. Accessing this data is only possible via the reflective EMF API, which is also implemented by generated code. While the interpreter uses only this API, it is completely transparent whether dynamic objects or generated code is used. Because no code generation of the meta models is required, meta models can be exchanged or modified easily and the code that uses it does not need to be adapted. This is even possible at runtime. An interesting observation of the performance evaluation is, that there is virtually no difference in execution times if the interpreter works on dynamic EMF objects compared to using generated code.

The interpreter is already used in several projects at our research group. Among them is a model transformation system based on Triple Graph Grammars [4] and a model-driven configuration management system [6]. The model-driven configuration management system uses the interpreter extensively, e.g., Story Diagrams are used to generate Story Diagrams. Thus, we are able to configure Story Diagrams at runtime, generate the configured Story Diagrams and instantaneously execute the configured Story Diagrams with the interpreter.

In this paper, we report on additional improvements and features that come with the new version of the Eclipse SDM tools. The Story Diagram interpreter and the graphical editor now provide an extension mechanism, that can be used to integrate interpreters and editing features for arbitrary textual expression languages. These can be used to express constraints or actions. At the moment, OCL is the only supported expression language. Furthermore, the interpreter natively supports EMF's map-typed references. These are comparable to qualified associations known from Fujaba and allow to define mappings between key and value objects. Native support for this feature improves performance because map-specific access operations are exploited. So-called Containment Links can now be used to model navigation paths between a container object and its direct and indirect elements. Here it is not necessary to know in advance how many levels of indirection lie between the container and the element. Of course, both kinds of links can also be used in the graphical editor. In addition to that, existing features like the pattern matching algorithm and the validation feature of the graphical editor were improved and extended. In the sequel of this paper, we briefly introduce our Eclipse SDM tools in Section 2. In Section 3, the improvements and new features of the interpreter and the graphical editor are described. The paper closes with an outlook on future work in Section 4, especially on the features that are currently missing.

2. ECLIPSE SDM TOOLS

Core of our Eclipse SDM tools is the Story Diagram interpreter for EMF-based Story Diagrams. It consists of several components: The *interpreter* itself, which is responsible for traversing the overall Story Diagram starting from the initial node and invoking the other components; the *variables manager*, which is the central store of variables and their values; additional *interpreters for expression languages*, that can be plugged into the Story Diagram interpreter via an extension mechanism to allow execution of expressions written in that language; and the *Story Pattern Matcher*, that executes a single Story Action Node. The pattern matcher tries to find matches for the Story Pattern using a dynamic pattern matching strategy, and creates and deletes objects if a match was found. The matching process is explained in detail in [5].

The interpreter provides a notification mechanism, which is based on EMF's notification mechanism. By attaching a notification listener to the interpreter and setting a debug flag, notifications about all relevant execution steps can be received. This allows for example to implement a visual debugger for Story Diagrams. In the context of a model-driven configuration management system [6], notifications were leveraged for incremental Story Diagram application. The notification mechanism is used to profile instances that were traversed during the Story Diagram application. If a Story Diagram is matched successfully, the traversed instances are stored together with the Story Diagram. Whenever these instances change, we know that the Story Diagram has to be applied again.

The interpreter is complemented by a graphical editor for Story Diagrams based on the Graphical Modeling Framework (GMF)[5]. Using GMF, graphical editors for EMF-based models can be created with comparably little effort. These editors already include basic features like Copy&Paste, printing or layouting.

Furthermore, the graphical editor allows to check a model for correctness and indicate modeling errors to the user. EMF provides an extensible validation mechanism for that purpose. Plug-ins can be integrated into this validation mechanism by realizing some extension points. They can register a validator for a specific meta model. If an instance of that meta model is validated, the validator is invoked. The validation can be triggered in the graphical editor (but also in the tree editor generated by EMF) or from Java code. Unfortunately, the validator has to be implemented in Java, which makes specifying simple validation rules rather cumbersome.

To ease the definition of validation rules, we use Xtend's Check[6] language which is a constraint language similar to OCL. Xtend provides a generic validator that can be used to execute Check constraints, which are declared in a text file. This allows to define the constraints, that should hold on a valid Story Diagram in a concise and easily maintainable way. An exception is the validation of textual expressions because this requires a more thorough analysis of the Story Diagram. For example, in order to check an OCL expression, the names and types of all available variables must be acquired first. This is very difficult to express in Check.

[5]http://www.eclipse.org/gmf/
[6]Xtend is part of Xpand: http://www.eclipse.org/modeling/m2t/?project=xpand

2.1 Expressions

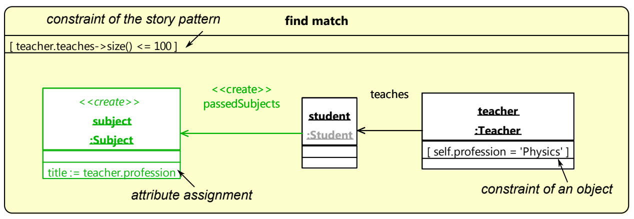


Figure 1: Expressions are used in many places to express constraints, queries and actions.

Story Patterns are a powerful mechanism to express graph patterns and graph transformations. However, some kind of constraints or actions can be expressed easier with a textual language. The interpreter and the Story Diagram editor provide an extension mechanism that allows other plug-ins to integrate textual expression languages. These plug-ins must provide at least an interpreter for their expression language. This interpreter is invoked by the Story Diagram interpreter. In addition, they can provide a custom *Source Viewer* implementation, that includes features like syntax highlighting and code completion (see Section 3.5 for more information). For OCL such an additional interpreter is already included. This is a new feature. Formerly, the evaluation of OCL expressions was directly integrated into the interpreter.

Fig. 1 shows several examples where constraints can be used. If they are used on a Story Pattern object (e.g., the *teacher*), the constraint is evaluated as soon as a possible match was found. This implies that other Story Pattern objects might not be matched at this time. Therefore, references to them should not be used in such constraints. Instead, these constraints should be attached to the overall Story Pattern, so they are evaluated after a match for the whole pattern was found. If used as constraints, expressions have to return a boolean value. However, they can also be used as queries to return an arbitrary value. This is the case with the attribute assignment of the *subject*. The expression is evaluated and its value is assigned to the *title* attribute. Within an expression, all created variables of the Story Diagram are available, e.g. the *teacher* in the attribute assignment of *subject*.

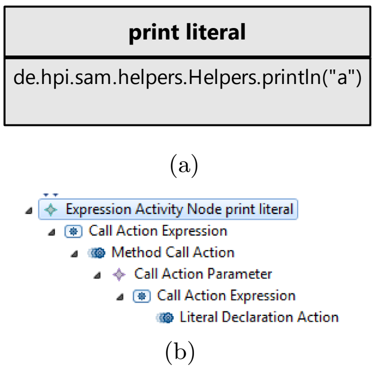

print literal
de.hpi.sam.helpers.Helpers.println("a")
(a)
Expression Activity Node print literal
Call Action Expression
Method Call Action
Call Action Parameter
Call Action Expression
Literal Declaration Action
(b)

Figure 2: An Expression Activity node that calls an external helper operation, in concrete (a) and abstract (b) syntax.

Another place are Expression Activity Nodes (see Fig. 2(a)), which are comparable to Fujaba's Statement Activities. They contain an expression, that is executed but whose return value is ignored. It is intended to use expressions with side effects here to perform arbitrary actions.

While OCL is free of side effects and the integration of existing scripting languages, like Lua, PERL, Python, etc. poses some difficulties, we decided to integrate another kind of expression language where the expressions are directly modeled as trees instead of a text string. Fig. 2(b) shows an example. These *Call Action Expressions* provide basic functionality to declare literals and variables, reference variables, create new objects and, most importantly, invoke arbitrary Java methods via reflection. *Call Action Expressions* can be nested and mixed with string expressions. In Fig. 2(b) a *Method Call Action* is used to invoke the method. The parameter *"a"* is provided by a *Literal Declaration Action*. A major advantage of *Call Action Expressions* is, that they can be evaluated very quickly. It is not necessary to parse a string. In contrast, using OCL expressions has a notable impact on the performance because they have to be parsed first.

3. NEW FEATURES OF THE ECLIPSE SDM TOOLS

In this section, the new features and improvements of the Story Diagram interpreter and graphical editor are presented.

3.1 Improved Matching Algorithm

The *Story Pattern Matcher* maintains lists of the bound and unbound Story Pattern objects, checked and unchecked links, and bound instance objects of a Story Pattern. When objects are bound or links are checked, objects are moved between these lists. Each time, after finding a match for a single Story Pattern object, these lists are copied and put on a stack. If later on a match turns out to be wrong, the stack is reduced. These copy operations are quite time consuming. Therefore, the algorithm was improved. Instead of copying the lists, transactions are created and put on a stack. Each time a match was found or a link was checked, the appropriate object is moved between the lists and a transaction is created. If the stack is reduced the transactions are simply rolled back. Furthermore, the lists were replaced by hash sets, which offer a higher performance especially for the *contains* operation. In a simple benchmark, these two improvements led to an increase of the pattern matching performance of about 100%.

3.2 Enhanced Story Patterns

A special feature of the Story Pattern Matcher is the analysis phase preceding the execution. Its purpose is to sort the links and objects and introduce new links into the Story Pattern to make the pattern matching process more efficient.

The introduction of new links is shown in Fig. 3. The *student* is a bound object (indicated by the grayed out type of the object). The pattern matching starts here to find matches for the other object. The *teaches* link leads from the teacher to the student but the reference in the meta model is a bidirectional reference. The interpreter notices that and creates an opposite link.[7] The same is done for containment references. EMF provides the *eContainer()* operation to get the

[7]Of course, the user can explicitly model both directions but the analysis phase makes this unnecessary.

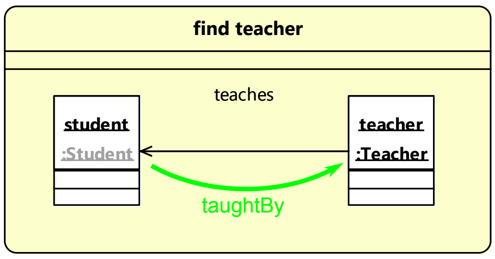
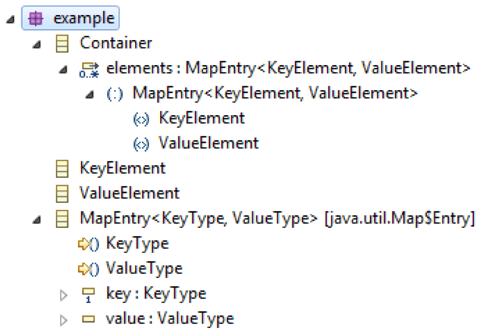


Figure 3: Another link is inserted that represents the opposite link of a bidirectional reference.

container object of an object. Suppose, the *teaches* reference is a unidirectional containment reference. The *teacher* can be easily matched by calling *eContainer()* on the student. Note, that these additional links are inserted at runtime and, therefore, are not visible in the graphical editor and thus to the user.

3.3 Map-Typed References



```
example
  Container
    elements : MapEntry<KeyElement, ValueElement>
      MapEntry<KeyElement, ValueElement>
        KeyElement
        ValueElement
  KeyElement
  ValueElement
  MapEntry<KeyType, ValueType> [java.util.Map$Entry]
    KeyType
    ValueType
    key : KeyType
    value : ValueType
```

Figure 4: A meta model containing a map entry.

Sometimes, mappings from keys to values are required to model qualified associations, for example. EMF provides so-called *Map-Typed References*, references to map entries, which are parameterized with key and value types. These references allow mapping keys to values. This is shown in Fig. 4. The class *Container* has a map, that maps *KeyElements* to *ValueElements*. To realize this, a helper class with instance type name *java.util.Map$Entry* must be created. It has a *key* and a *value* attribute and generic type parameters. The *elements* reference of the *container* is a to-many containment reference to the *MapEntry*, which is parameterized with the *KeyElement* and *ValueElement* types. EMF's code generator recognizes this pattern and creates a map instead of a list.

This can be modeled in the same way in a Story Pattern, which is shown in Fig. 5. The map entry is a distinct object in the Story Pattern. The *value* object can be easily obtained by querying the map because the *key* object is already bound, here. However, now the interpreter does not exploit the fact that this is a map. The interpreter rather treats it as a list of map entries, i.e. it examines all map entries and checks for each entry whether its key is the *key* object. Furthermore, the *MapEntry* object is not relevant to the user of the Story Diagram. It rather makes the diagram more complicated. Therefore, we introduced a special kind of link, a *Map Entry Story Pattern Link*.

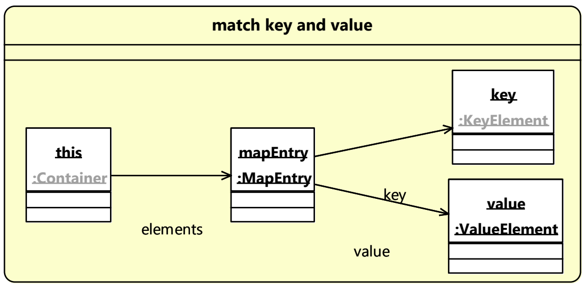


Figure 5: Modeling a map entry as a distinct object in the Story Pattern.

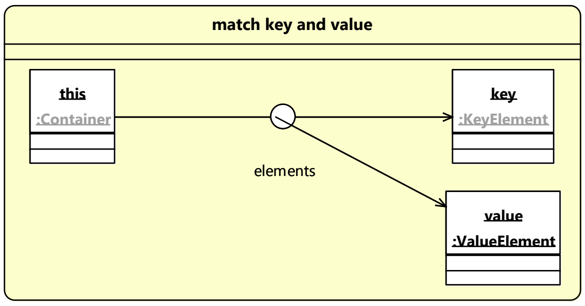


Figure 6: Modeling a map entry as a *Map Entry Story Pattern Link*.

Fig. 6 shows the same case like Fig. 5 but this time the map entry is hidden by a *Map Entry Story Pattern Link*. When this type of link is used, the interpreter will exploit the fact, that there is a map and simply query it for the value belonging to the *key* object. Of course, this is not the best possible solution because it is a work around. A better way would be if EMF would provide a more elegant way to model maps.

3.4 Containment Links

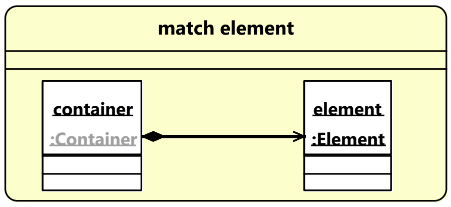


Figure 7: A Containment Link indicates that the *Element* should be contained directly or indirectly in the *Container*.

Ordinary Story Pattern links specify which reference from the meta model they are representing. So the interpreter only needs to follow the corresponding instance links to find matches for unbound objects. However, sometimes it is not known how the unbound objects are connected to a bound root object. It may be unknown, via which reference they are connected or how many levels of indirection are in between. If it is at least known that objects are connected via containment associations, *Containment Links* (see Fig. 7 can be used to model this case. The *Containment Link* indicates that the *Element* is directly or indirectly contained in the *Container*. This feature can be implemented very easily by using the *eAllContents()* operation of EMF's reflective API. This operation returns a tree iterator, that returns all direct and indirect children. The opposite direction is also possible, i.e. the element is bound and a match for the container is sought. In this case, the interpreter climbs the containment hierarchy upwards by calling *eContainer()* repeatedly until a match was found or the top container was reached.

3.5 Story Diagram Editor Improvements

A major improvement of the graphical Story Diagram editor is the support for code completion and syntax highlighting for textual expression languages. This is useful for users that are unfamiliar with the syntax of a certain language. Technically, this feature is realized by providing custom implementations of *org.eclipse.jface.text.source.ISourceViewer* and implementing a small interface that allows the graphical editor to access this custom *ISourceViewer* and get and set the text of the viewer. A custom implementation allows to provide syntax highlighting and other visual annotations, code completion and content assist features. For OCL, we provide basic syntax highlighting and code completion. If a plug-in does not provide an own implementation, a default source viewer is used. Of course, advanced editing features are not available in this case.

Apart from that, the graphical editor now supports all the new types of links presented in the preceding sections. Furthermore, the validation rules were largely extended to detect a lot more modeling errors. These extensions stem mostly from experience gathered while using the graphical editor. Also a simple launch dialog was added, so the user can directly execute a Story Diagram. Only a meta model, an instance model (in form of an XMI file), and the Story Diagram are required.

4. OUTLOOK

The interpreter solves our two most urgent problems. The dynamic pattern matching strategy leads to a better average and worst-case performance. The interpreter offers a tight integration with EMF and is compatible to dynamic EMF, which improves the flexibility of Story Diagrams and their application. The additional features of the interpreter in combination with the graphical editor and its features improve the usability of Story Diagrams within Eclipse, especially for people unfamiliar with Story Diagrams.

However, there are still limitations and space for improvements. A major disadvantage of the interpreter, compared to Fujaba, is the lack of integrating arbitrary Java code directly into Story Diagrams, because no interpreter is available. Currently, we employ expression languages to overcome this drawback. These are either textual languages or basic tree based models (*Call Action Expressions*, see Section 2.1), that can be used to express conditions and queries, and perform actions. However, OCL is the only supported textual language, yet. Another limitation is that Story Diagrams need to be linked into an operation definition of a class of a meta model. This limits the dynamic capabilities because the operation signatures must be defined a-priori and are thus fixed. Therefore, we plan a less restricted concept. The idea is to hand over a dynamic start-graph, which does not need to fit into the signature of the related method. In addition, we plan to extend the interpreter with basic features known from Fujaba. For example, negative application conditions are not supported, yet. Instead, OCL constraints have to be used for this purpose. Other missing features are path expressions, optional objects, sets of objects and syntax checks of OCL expressions. Another problem is the mentioned impact of evaluating OCL expressions on the performance. A possible solution could be to parse the expression beforehand and store the expression's abstract syntax tree in the Story Diagram.

Acknowledgements

We like to thank all students that helped implementing and improving the Story Diagram interpreter and the graphical editor. These are in particular Mark Liebetrau, Sebastian Wätzoldt and Christian Lück.

5. REFERENCES

[1] B. Becker, H. Giese, S. Hildebrandt, and A. Seibel. Fujaba's Future in the MDA Jungle - Fully Integrating Fujaba and the Eclipse Modeling Framework? In *Proceedings of the 6th International Fujaba Days*, 2008.

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

[3] L. Geiger, T. Buchmann, and A. Dotor. EMF Code Generation with Fujaba. In *5th International Fujaba Days*, Kassel, Germany, October 2007.

[4] H. Giese and S. Hildebrandt. Efficient Model Synchronization of Large-Scale Models. Technical Report 28, Hasso Plattner Institute at the University of Potsdam, 2009.

[5] H. Giese, S. Hildebrandt, and A. Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. In T. Magaria, J. Padberg, and G. Taentzer, editors, *Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, 2009.

[6] H. Giese, A. Seibel, and T. Vogel. A Model-Driven Configuration Management System for Advanced IT Service Management. In *Proceedings of the 4th International Workshop on Models@run.time at the 12th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, Denver, Colorado, USA, October 2009. accepted.

[7] J. Johannes. Letting EMF Tools Talk to Fujaba through Adapters. In *Proceedings of the 6th International Fujaba Days 2008*, 2008.

[8] T. Klein, U. A. Nickel, J. Niere, and A. Zündorf. From UML to Java And Back Again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, 1999.

Specification of Triple Graph Grammar Rules using Textual Concrete Syntax

Mirko Seifert
Lehrstuhl Softwaretechnologie
Fakultät Informatik
Technische Universität Dresden
Dresden, Germany
mirko.seifert@tu-dresden.de

Christian Werner
Lehrstuhl Softwaretechnologie
Fakultät Informatik
Technische Universität Dresden
Dresden, Germany
s7436532@inf.tu-dresden.de

ABSTRACT

Triple Graph Grammars provide a powerful mechanism to specify bidirectional model transformations. Complex transformations or synchronisation scenarios can be described using declarative rules. However, the specification of these rules is often hard, because rules are specified at the abstract syntax level of the involved models. Furthermore, rules are mostly visualised and edited graphically. This is very feasible for modelling languages that have a graphical syntax, but for textual modelling languages a gap between the languages and the rules is introduced. In addition, large rules written in graphical syntax can easily become confusing and hard to read.
To tackle these problems, we propose to automatically extend the textual syntax for the involved models and use it to specify rules. We explore the benefits and drawbacks of rules that are based on concrete textual syntax.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General

General Terms

Design

Keywords

triple graph grammars, model transformation, textual syntax

1. INTRODUCTION

Triple Graph Grammars (TGGs) [14] have been first introduced by Andy Schürr in 1994 as an extension to pair grammars. Various research papers have shown their applicability in different domains. For example, TGGs have been employed to perform model transformations [11], tool integration [12], and incremental model synchronisation [4]. TGGs were not only successful from an academic perspective, but have also started to gain more widespread use in the industry. This is mainly reflected by the Query View Transformations (QVT) [13] standard which shares concepts with TGGs (see [5] for a detailed comparison).

Current implementations (e.g., Fujaba[1] or MOFLON[2]) provide facilities to specify TGG rules and execute them. Besides debugging, the specification of these rules is the most complicated part when setting up a new TGG transformation. Given that a rule designer is familiar with the concepts of TGGs and in particular the semantics of the different elements of a rule (e.g., *required* and *create* nodes and links), the specification is still not straight forward.

From our perspective, one of the main issues is, that rules are defined at the level of abstract syntax. Instead of using the concrete syntax of the models that are transformed or synchronised, rule designers must refer to the abstract elements of the models (i.e., the meta classes). While this may not sound so complicated at first glance it can turn out to be quite cumbersome. Tiny bits of concrete syntax do often correspond to a large graph if represented in abstract syntax. A lively example are abstract representations of expressions, where simple terms turn into large expression trees.

In addition, rules are usually expressed using graphical syntax. While this has clear advantages in some cases it can imply problems in others. If TGGs are used to transform models that have a textual syntax on their own (e.g., textual Domain-Specific Languages (DSLs)), the graphical rule syntax creates a gap between the subject and the specification of the transformation. Transformation designers are intrinsically familiar with the concrete syntax of their languages. Thus, rule specification gets easier the closer it is to the subject languages.

Being at the heart of TGG-based model transformation and synchronisation, the rule specification is an important issue to pursue the adoption of the TGG formalism. In this paper, we will therefore present how the concrete textual syntax of modelling languages can be used to automatically obtain a TGG rule specification language having a syntax that is close to the original one. The approach will be presented based on a running example (Sect. 2). After explaining the syntax extension (Sect. 3) an explanation of the rule extraction follows (Sect. 4). We compare our work with related publications in Sect. 5 and draw conclusions in Sect. 6.

2. RUNNING EXAMPLE

Within this paper we will use an example from [10] to illustrate our ideas. The example involves a transformation between petri nets and toy train models. The petri nets consist of nodes, arcs and tokens. Nodes can be either transitions or places. Tokens are assigned to a place. The corresponding meta model is shown in Fig. 1.

[1]http://www.fujaba.de
[2]http://www.moflon.org

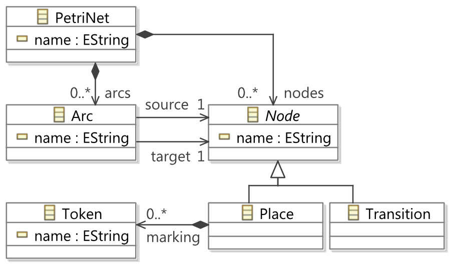


Figure 1: Meta Model for Petri Nets

The toy train models (cf. Fig. 2) are called projects and contain components. Components can be either switches or tracks. Connections can be drawn between ports, where each component can have multiple in and out ports. Trains are assigned to a component denoting that the train is currently located on this track or switch.

Even though the graphical notation of petri nets is used more widely, we will express instances in textual syntax. For the toy train models, being a typical example for DSLs, the textual syntax may be a good choice, because it is easy to define and tool support can be generated automatically. Other transformations that involve textual languages (e.g., programming languages such as Java) are most suitable for the textual rule specification. Nonetheless, we will use the simple transformation between petri nets and toy trains to sketch our general ideas.

Note that textual syntax should not be considered as a contradiction to graphical syntax here. Rather, it is an alternative one. Both types can be used in combination, for example to version control or exchange models in textual form (compare [1]) and edit them in graphical syntax.

Figure 3 shows two example models. To define these syntaxes and to generate tool support we used EMFText [8], but any other textual concrete syntax mapping tool would suffice too. The two example languages are available from the EMFText Syntax Zoo[3].

The transformation between these two models will map the

[3]http://www.emftext.org/zoo

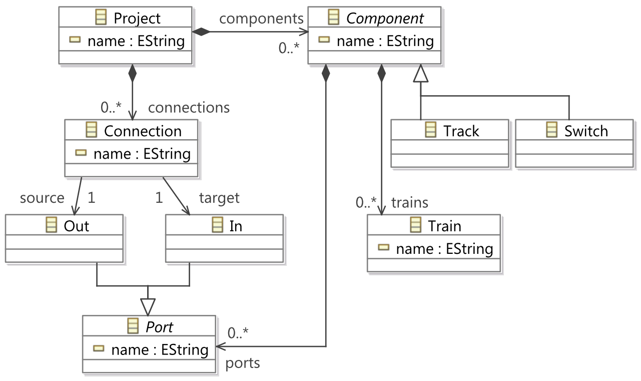
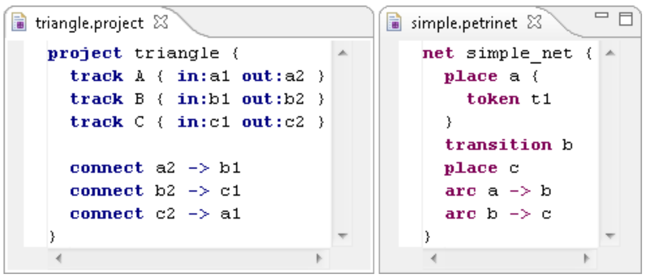


Figure 2: Meta Model for Toy Trains



```
project triangle {
  track A { in:a1 out:a2 }
  track B { in:b1 out:b2 }
  track C { in:c1 out:c2 }

  connect a2 -> b1
  connect b2 -> c1
  connect c2 -> a1
}
```

```
net simple_net {
  place a {
    token t1
  }
  transition b
  place c
  arc a -> b
  arc b -> c
}
```

Figure 3: Example Toy Train Project and Petri Net in Concrete Syntax

toy train models to their respective dynamic semantics expressed in terms of a petri net. The rules that specify this transformation have been taken from [10]. Before we show what the textual variants of these rules look like, we shortly sketch how the textual syntaxes of the involved languages can be automatically extended to obtain the rule syntaxes.

3. LANGUAGE EXTENSION

One of the problems mentioned before is that TGG rules are usually specified at the level of abstract syntax. To allow for a more intuitive specification, we want to use the concrete textual syntax of the languages involved in the transformation. More specifically, we want to extend the involved languages with annotation facilities to enable the specification of rules. The general approach used to achieve this is shown in Fig. 4.

Basically the meta models of both languages involved in a transformation are extended with annotation concepts defined in an annotation meta model. Instances of the enriched meta models can contain annotations, which are evaluated by a rule derivation algorithm. Then, a TGG rule is created from each pair of models. As the model instances shall be written in concrete textual syntax, the syntax for both languages must be also extended (not shown in Fig. 4). One may also consider to use annotation mechanisms that are already available in the involved languages. However, this restricts the rule specification to languages, which do have such mechanisms, whereas our approach is applicable for every language.

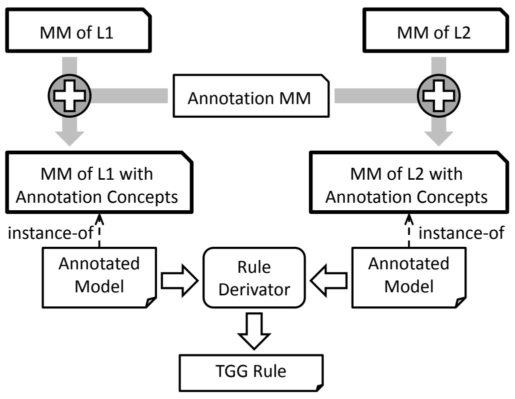


Figure 4: Rule Specification based on Language Extension

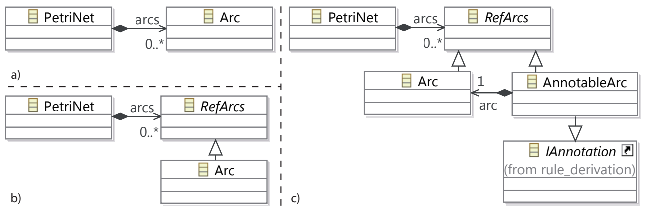


Figure 5: Excepts from the original (a), the extensible (b) and extended Meta Model (c) for Petri Nets

The annotations added to the meta models, are specific to rule specification. For example, annotations tag elements that correspond to each other. Elements must also be typed as being required or created. Furthermore, constraints that control the rule application need to be expressed.

3.1 Meta Model Extension

To extend a meta model it needs to be either designed for extensibility or, if this is not the case, refactorings can be applied. In [9] details about the requirements for extensible meta models and the respective refactorings can be found. The paper refers to Ecore models, but the principles can be applied to other metamodelling languages as well.

Briefly said, the extensibility is established by using dedicated abstract classes to type each reference. By doing so, new types can be attached to the reference by extension of these abstract classes. In addition, all attributes having primitive types are replaced by type wrappers. This wrapping is needed to raise primitive types to the level of complex types. Thus, the same mechanism (references with abstract type) can be used to allow for arbitrary extensions.

As an example, consider the (non-extensible) meta model shown in Fig. 1. Here, the extensibility can be established by replacing the reference `arcs` between `PetriNet` and `Arc` with a reference to a new abstract class `RefArcs`, being a superclass of `Arc`. By doing so, new subclasses of `RefArcs` can be added, which allows arbitrary extensions.

Figure 5 shows an excerpt from the meta model before and after this refactoring (Note: attributes were omitted). In the original meta model (a) the reference `arcs` directly refers to class `Arc`, whereas in the extensible version (b) the reference refers the the abstract class `RefArcs`. In (c) an new subclass `AnnotableArc` is introduced, which contains an ordinary arc and inherits from the annotation concept `IAnnotation`. This class is imported from our TGG annotation meta model.

We will use these annotations to add information to models that allow the derivation of rules. Annotations do have identifiers and a type, which is shown in Fig. 6.

When specifying rules, every meta class of the involved languages can potentially be annotated. Thus, all meta classes need to be extended to support our rule annotations. Therefore, for each existing meta class the extension sketched in Fig. 5 is performed. A new subclass is created that inherits both from the reference type and `IAnnotation`. Having done this for both meta models, we obtain support for annotations. Note that this extension can easily be automated.

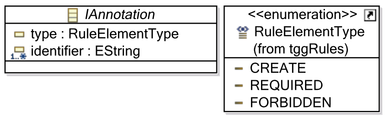


Figure 6: Meta Model for Annotations

3.2 Syntax Extension

After extending the meta models to capture rule annotations, the concrete syntax must be extended along the same path. For each new meta class appropriate syntax must be defined. As most other textual syntax tools, EMFText allows to define syntax per meta class. Thus, to obtain a syntax definition for the meta model with annotation support, the original syntax is imported and new rules, one for each new meta class, are added. The new syntax rules basically prefixes the original syntax (e.g., the one defined for class `Arc`) with syntax elements for the attributes of class `IAnnotation` as shown in Listing 1.

```
Arc          ::= "arc" (name[IDENT])?
                 source[IDENT] "->" target[IDENT];
AnnotableArc ::= (identifier[IDENT])+
                 (type[TYPE])? arc;
```

Listing 1: Syntax Rules for Arc and AnnotableArc

The syntax extension is identical for all new meta classes and can thus be also performed automatically. Equipped with two extended meta models and two matching syntax definition, EMFText can be used to generated tool support (e.g., parsers and editors) for the two new languages. To summarise, we can automatically extend both the meta model and the syntax of the languages involved in a transformation and generate tool support. Thus, one can start specifying rules in concrete textual syntax without any manual effort.

4. RULE DERIVATION

Up to now, we have shown how to automatically derive languages with annotation support. We have also sketched, what these annotations look like, but the more interesting issue is how to derive TGG rules from annotated models. Before giving details of the rule derivation lets shortly recapitulate the elements of TGG rules.

TGG rules have a left-hand and a right-hand side. Both sides are graphs and therefore consist of nodes and links. Nodes can be either correspondence nodes or model element nodes (i.e., they refer to a meta class of one of the involved languages). Links connect nodes within rules. Since TGG rules are usually non-deleting, the two sides are often merged into one graph by tagging the new nodes (i.e., the ones that are present at the right-hand side only) as *create* nodes, which is often denoted by `++`. All other nodes are marked as *required* nodes. In addition TGG rules can contain constraints, assignments or *forbidden* nodes. However, for the scope of this paper, we will not deal with these concepts.

We derive one rule from each pair of annotated models. One could also consider, the rule derivation as abstraction, because some parts of the pair are not included in the rule. To perform rule derivation, we execute the following steps:

- For each annotated model element, create a rule node
- For each set of model elements that are annotated with the same identifier, create a correspondence node and create links connecting the new correspondence node with the respective rule nodes
- Mark all rule nodes as *create* where the corresponding model element is annotated as *create* element
- For each pair of model elements that is connected by exactly one reference create a link between the respective rule nodes
- For each pair of model elements that is connected by multiple references use the references specified in the annotation and create links between the respective rule nodes

To give an example for the rule derivation process, consider Fig. 7 (taken from [10]). The rule specifies how to map tracks to petri net components. Each track is mapped to an arc, its `in` port is mapped to a place, while its `out` port is mapped to a transition in the petri net. The prerequisite for applying this mapping is an existing correspondence between the project and the petri net.

The very same rule can be specified in textual syntax using the pair of models shown in Fig. 8. The correspondence nodes are highlighted in bold font face. Black font indicates required correspondence node, whereas create nodes a shown in green font and followed by `++`.

By looking at Fig. 8 it is easy to see, how the nodes in the textual syntax and the nodes in the graphical representation relate to each other. Basically each box (in the graphical syntax) can be found as an annotated element (in the textual syntax). For example, `@Pr2PN project` is the annotated node that triggers the creation of the `:Project` node in the upper left corner of the TGG rule in Fig. 7. The same applies to `@Pr2PN net`, which causes the creation of the `:PetriNet` node. The annotation `Pr2PN` itself is transformed to the correspondence node `:Pr2PN` and the two links connecting `:Pr2PN` with `:Project` and `:PetriNet`.

This part of the rule derivation is performed by steps 1 and 2. Applying both steps to all other annotated elements (i.e., the ones tagged with `Cp2PN`, `IP2Pl`, and `OP2Tr`) yields all boxes contained in the rule shown in Fig. 7. In addition, links between the correspondence nodes and the domain nodes have been created. However, the nodes are not yet typed

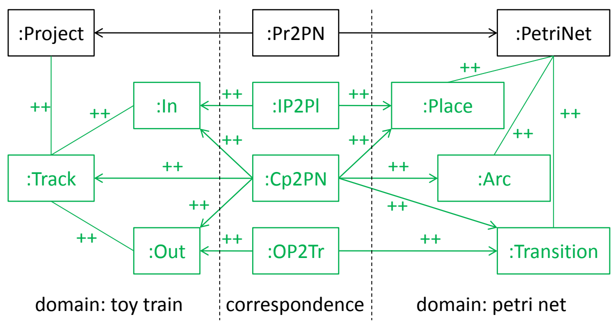
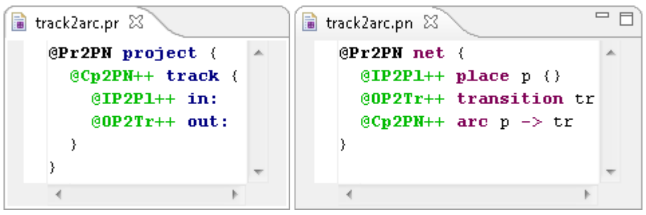


Figure 7: Visual Rule for Mapping Tracks



```
@Pr2PN project {
  @Cp2PN++ track {
    @IP2Pl++ in:
    @OP2Tr++ out:
  }
}
```

```
@Pr2PN net {
  @IP2Pl++ place p {}
  @OP2Tr++ transition tr
  @Cp2PN++ arc p -> tr
}
```

Figure 8: Textual Rule for Mapping Tracks

as required or create. This is done in step 3 of the rule derivation procedure using the type of the annotation.

Now, step 4 takes care of the links that connect elements in a single domain. For example, the link between `:Project` and `:Track` has not yet been established. To obtain these links, the rule derivation must analyse the relations between the elements in each textual model. If model elements are connected by references, links need to be created in the TGG rule. If elements are connected by a single reference only, one rule link is obtained. If elements are connected by multiple references the rule must be refined to specify which references are important for rule derivation. This may be either all references or only selected ones (step 5).

As mentioned in the introduction, we do not handle constraints on attributes yet. Possibly, one could automatically create equality constraints if attribute values (e.g., the names) of elements match. However, this not feasible for all types of attributes. For string-typed attributes many possible values exist, which makes it easy to choose two equals values (e.g., names). For boolean attributes the case is different, because there are only two possible values. Thus, wrong constraints may be derived, because values match even though this is not meant to express equality. Evaluating this is subject to future work.

5. RELATED WORK

Visual rules were already used to specify transformations in the early papers that introduced TGGs [14]. Later, different notations based on object-diagrams were presented that integrated more smoothly with UML modelling [6]. However, the specification was still based on abstract syntax. While applying TGGs to more and more problems, the difficulty of specifying TGG rules (complexity, readability, reuse) has been discovered and studied. In [2], *Triple Patterns* a compact and visual notation has been proposed. The authors share the idea of a more compact representation of TGG rules, but use a visual specification which is still based on abstract syntax. This is in contrast to our work, which focuses on textual concrete syntax.

Other works, that are based on concrete syntax [15, 3] target specification by example. Here, the syntax of the involved languages can be used to specify pairs of models. Based on these pairs, the goal is to derive TGG rules that match the transformation semantics stated by the two models. In other words, rules are derived, which produce one of the models if using the other one as input. This approach is similar to ours regarding the use of concrete syntax, but differs in the way rules are defined. While we explicitly state the correspondence part of the rules using annotations, specification by example approaches try to derive this part automatically. While this may in principle yield the same result, more example pairs may be needed in the latter case. In addition, transformation designers need to check the derived rules to make sure their expectations are met, while our approach allows to state the expected correspondences beforehand.

6. CONCLUSIONS

In this paper we have shown how to use textual syntax to specify TGG rules. Based on the concrete syntax of the involved languages, we derived extended languages providing annotation support. This extension can be performed fully automatic, which enables the application of our approach to arbitrary textual modelling languages. The gathered annotation support was then employed to add additional information to textual models. Namely, the correspondence between two models was expressed. In contrast to specification by example, which does also use pairs of models to derive rules, we do explicitly state how models correspond instead of deriving this relation.

In the course of this work we came to the conclusion that rule specification based on annotations should be applied to graphical syntaxes as well. By doing so, one could annotate the models in their natural syntax. For example, rules that involve both a graphical language and a textual one (e.g., UML to Java transformations) could be specified using annotations in the graphical syntax (on the UML side) and in the textual syntax (on the Java side). This would allow to use the concrete syntax of the involved languages instead of operating at the abstract syntax level. However, the specification and extension of graphical syntax is not understood as well as its textual counterpart.

The work presented here, yields different open questions. First, the completeness of the rule derivation has to be shown. It is not yet clear, whether all meaningful TGG rules can be specified using the presented annotations. We managed to specify all the basic rules given in [10], but there may be other rules, that can not be specified yet. However, the example rules indicate that the annotation-based specification can be very compact and easy to read if textual languages are subject to TGG transformations. Second, more transformation scenarios need to be conducted to evaluate the practical applicability of our approach. In particular the integration into one of the tools supporting TGGs could give more insights about annotation-based rule derivation.

Acknowledgments

This research has been co-funded by the European Commission within the FP6 project MODELPLEX #034081 and by the German Ministry of Education and Research within the project SuReal. We are also very grateful to Jendrik Johannes who encouraged this work and contributed a lot of ideas in our discussions.

7. REFERENCES

[1] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In *Applications and Theory of Petri Nets 2003: 24th International Conference*, pages 1023–1024, Eindhoven, The Netherlands, June 2003.

[2] J. de Lara, E. Guerra, and P. Bottoni. Triple Patterns: Compact Specification for the Generation of Operational Triple Graph Grammar Rules. In K. Ehrig and H. Giese, editors, *GTVMT'07*, volume 6, 2007.

[3] I. García-Magariño, J. J. Gómez-Sanz, and R. Fuentes-Fernández. Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages. In R. F. Paige, editor, *ICMT*, volume 5563 of *LNCS*, pages 52–66, 2009.

[4] H. Giese and R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In Hartman and Kreische [7], pages 543–557.

[5] J. Greenyer and E. Kindler. Reconciling TGGs with QVT. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *LNCS*, pages 16–30. Springer, 2007.

[6] L. Grunske, L. Geiger, and M. Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. In Hartman and Kreische [7].

[7] A. Hartman and D. Kreische, editors. *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings*, volume 3748 of *LNCS*. Springer, 2005.

[8] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and Refinement of Textual Syntax for Models. In *Proceedings of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA)*, 2009.

[9] F. Heidenreich, J. Johannes, M. Seifert, C. Wende, and M. Böhme. Generating Safe Template Languages. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE'09)*, 2009.

[10] E. Kindler and R. Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report tr-ri-07-284, University of Paderborn, D-33098 Paderborn, Germany, June 2007.

[11] A. Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego Bay, Jamaica*, 2005.

[12] A. Königs and A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. *Electronic Notes in Theoretical Computer Science*, 148(1):113–150, 2006.

[13] OMG. MOF Query/View/Transformation Specification Version 1.0. Technical report, Object Management Group (OMG), 2008.

[14] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Proceedings of 20th International Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany*, volume 903 of *LNCS*. Springer Verlag, 1994.

[15] D. Varró. Model Transformation by Example. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *LNCS*, pages 410–424. Springer, 2006.

Improving Dynamic Design Pattern Detection in Reclipse with Set Objects

Markus von Detten, Marie Christin Platenius
Software Engineering Group, Department of Computer Science,
University of Paderborn, Paderborn, Germany
[mvdetten|mcp]@mail.uni-paderborn.de

ABSTRACT

Design pattern detection is a reverse engineering methodology that helps software engineers to analyze and understand legacy software by recovering design decisions and thereby providing deeper insight into software. Recent research has shown that a combination of static and dynamic source code analysis can produce better results than purely static approaches. In this paper we present an extension of the pattern detection approach proposed by Wendehals [22]. In particular, we extend the specification language for behavioral patterns to increase its expressiveness and the approach's recall by introducing the concept of set objects.

1. INTRODUCTION

Due to requests for new features and the discovery of defects, software has to be continuously adapted and maintained during its life cycle. Incomplete documentation or the unavailability of the original developers often complicate this task and are among the reasons that software engineers often spend more time to maintain a complex software system than to actually develop it. According to Sommerville 50% to 75% of the total programming effort spent on a system are devoted to maintenance [17].

The tedious and error-prone task of understanding a large system can be supported and simplified by reverse engineering tools that recover the design of the software and try to locate the application of design patterns. Identifying these pattern instances can help the reverse engineer to quickly understand a software system and thereby speed up the maintenance process. Design patterns were first introduced by Gamma et al. and represent good solutions to frequently occurring problems in object-oriented software design [6]. Since then design patterns have been thoroughly researched and their detection for reverse engineering purposes has been the subject of many scientific publications (e.g. [1, 2, 7, 8, 9, 11, 12, 15, 16, 18]).

One of the main challenges in design pattern detection lies in achieving a high precision and recall, i.e. in finding the actual pattern implementations in the software while avoiding false positives. Especially the existence of many implementation variants for the various patterns leads to incorrect or incomplete detection results.

Lothar Wendehals presented an approach that combines static and dynamic analysis to reduce the number of false positives by taking the runtime behavior of the analyzed software into account [19, 22]. In this paper we present an extension of his approach that aims at increasing its recall by making the pattern specification language more expressive. For this we introduce set objects and each fragments into the specification language and adapt the analysis process accordingly.

The remainder of this paper is organized as follows: First, we give a general overview of the pattern detection process by Wendehals. In Section 2 we present an example and use it to demonstrate the shortcomings of the pattern specification language. A suitable extension of the language is proposed in Section 3. Section 4 deals with the realization of the approach that is then evaluated on a real software system in Section 5. In the subsequent Sections we discuss related work, draw conclusions and sketch ideas for future work.

2. STATIC AND DYNAMIC DESIGN PATTERN DETECTION

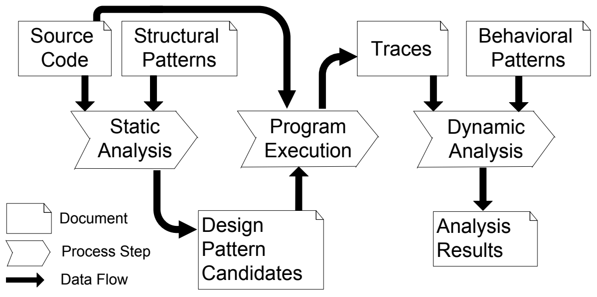


Figure 1: Static and dynamic analysis [22]

There are many examples in literature that use only a static source code analysis to recover design patterns (e.g. [1, 9, 15, 16, 18]). Although a static analysis is not limited to considering structural properties of the software under analysis certain object-oriented concepts like polymorphism and dynamic method binding make a precise static behavior analysis impossible. Hence, a common drawback of these approaches is that they generate false positives when design patterns have a similar structure. The *State* and *Strategy* patterns [6] are good examples for this: Their static structure is identical and they differ only in their runtime behavior. Common static pattern detection approaches often recognize implementations of the *State* pattern also as *Strategy* pattern implementations and vice versa. Obviously one of those results is always a false positive.

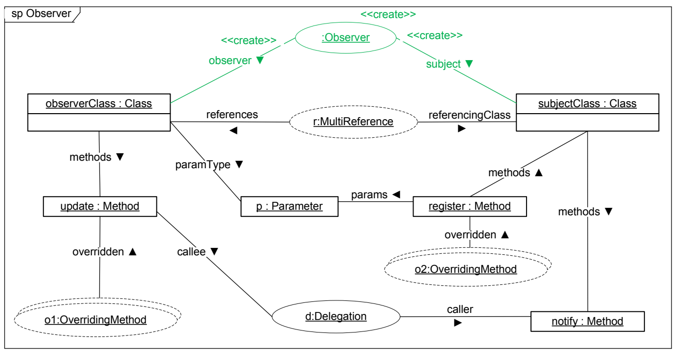


Figure 2: *Observer* structural pattern

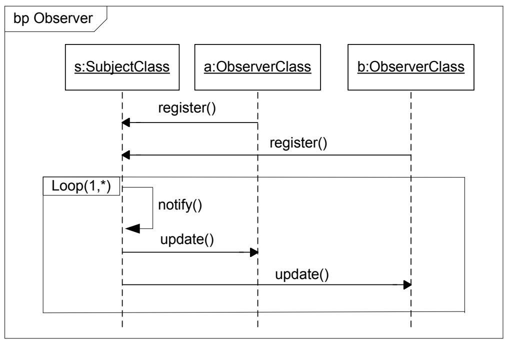


Figure 3: *Observer* behavioral pattern

Figure 1 shows the pattern detection process as proposed by Wendehals [22]. It uses the source code of the software system and a library of structural patterns to carry out a static analysis. In his approach, that builds on the work in [10] and [11], graph grammar rules are used for the specification of structural patterns (cf. Section 4). The result of the static analysis is a set of possible implementations of design patterns, the so-called *pattern candidates*. Pattern candidates are sections in the source code whose structure corresponds to the structural patterns used in the analysis. Due to structurally similar patterns, the result set may contain many false positives. At this point a dynamic analysis is used to confirm or reject the candidates.

After detecting the pattern candidates, the software system under analysis is executed manually and the candidates' behavior is traced. Depending on how often a candidate's classes are instantiated during execution time, a number of traces is generated for each candidate. The candidates' expected behavior is described with behavioral patterns based on UML 2.0 sequence diagrams [13]. During dynamic analysis the traces are compared with the corresponding behavioral patterns. If the majority of a candidate's traces match the behavioral pattern, it is likely that the candidate is an actual design pattern implementation and the candidate is confirmed. If most of the traces for a candidate do not match the behavioral pattern, it probably is a false positive and thus rejected.

2.1 Example

Throughout this paper we use the *Observer* pattern to explain the pattern detection approach devised by Wendehals and our extension. Gamma et al. describe the *Observer* pattern's intent as follows:

> "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." [6]

Figure 2 shows the structural description of the *Observer* pattern in the notation introduced in [11] and [12]. There are two classes `subjectClass` and `observerClass`. The subject class has the methods `register`, which takes an observer object as parameter, and `notify`[1]. The observer class has an `update` method. The ellipses are so-called *Annotations* and refer to subpatterns which are specified in other diagrams. Annotations represent instances of required subpatterns. In this case the annotations indicate that the subject class' notify method must implement a delegation to the observer class' update method. The methods update and register should be overidden. (The shown observer and subject classes are intended to be subclassed by concrete observers and subjects that implement their own specific behavior.) The dashed lines of the `OverriddingMethod` annotations indicate that these subpatterns are not mandatory for the detection of the *Observer* pattern. The `MultiReference` annotation expresses that a subject references arbitrarily many observers. Finally, the `Observer` annotation that is marked with `create` is created when the depicted structure is found in the software system under analysis. It tags the structure as candidate for the *Observer* pattern.

[1]The object names in the pattern are only variables that are matched to real names during the pattern detection process.

Figure 3 illustrates the expected behavior of the *Observer* pattern in the syntax defined in [22]. It shows one object `s` of the type `subjectClass` and two observer objects `a` and `b` of the type `observerClass`. The types refer to object names from the structural pattern (cf. Figure 2). `a` and `b` both call the register method of the subject class to register themselves for the subject's updates. The following loop fragment indicates that the enclosed message sequence must occur at least once but can occur arbitrarily many times. It states that whenever the subject class calls its notify method each observer's update method has to be called. If an observer candidate fails to show this behavior it probably is a false positive (or a variation of the design proposed in [6]).

For details on the specification of behavioral patterns with sequence diagrams we refer to [20]. More on the dynamic pattern detection algorithm can be found in [23].

2.2 Shortcomings of the Approach

One problem of the described approach lies in the use of absolute quantities of objects where in reality arbitrarily large sets of objects can participate in the pattern. The behavioral pattern in Figure 3 uses an exemplary situation with two observer objects to specify the desired behavior of an implementation of the *Observer* pattern. In reality any number of observer objects could communicate with the subject as long as the messages occur in the correct order. The behavioral analysis algorithm is currently limited to only recognize traces with exactly the specified number of objects as correct. In the example only candidate instances with two observer objects would be deemed accurate. Traces that represent the same situation with any other number of objects and otherwise conform to the behavioral pattern are rejected. This increases the probability that a candidate is incorrectly labeled as a false positive.

The same problem arises for all other patterns that involve a possibly arbitrarily large set of objects. Examples are the *State* pattern (an object can be in one of arbitrarily many states) and the *Chain of Responsibility* pattern (a request is passed down an arbitrarily long chain of handler objects until one handler consumes it) [6].

3. EXTENDING BEHAVIORAL PATTERNS WITH SET OBJECTS

Our approach solves the mentioned shortcomings by introducing a new element to the behavioral pattern specification language: the *Set Object*. A set object represents an arbitrarily large set of objects of the same type. With this new construct the *Observer* pattern can be specified without the need to predefine the exact number of observer objects involved at execution time.

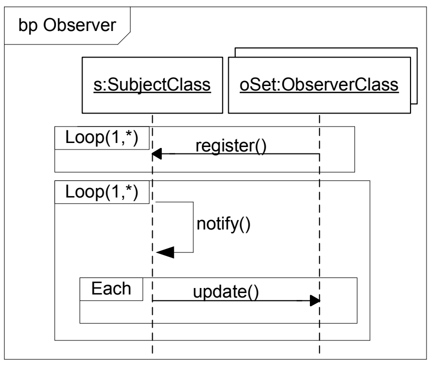


Figure 4: *Observer* behavioral pattern with set object

Figure 4 shows the *Observer* behavioral pattern with the new element. The set object is depicted by the double border and replaces the two separate `observerClass` objects from Figure 3.

The introduction of the set object necessitates proper semantics for messages between set objects and regular objects. A message from a set object to a regular object represents a method call from *one* of the objects in the set to the regular object. An example for this is the `register` message in Figure 4. Analogously, a message in the opposite direction represents a call to *one* object that is of the same type as the set object.

Additionally, we need to model the case that a method is called on *each* object in a set. For this we introduced a new combined fragment, the *Each Fragment*, which has semantics similar to the loop fragment. An each fragment can only be used for messages from regular objects to set objects (or vice versa) and means that the contained messages are sent to each object in the set (or from each object in the set to the regular object). Note that the semantics for message passing from one set object to another in conjuction with an each fragment remains undefined here. If such a construct was allowed, it would have to be specified if a message should be passed from each object in set 1 to each object in set 2 or if other combinations would be appropriate. However, in our investigations we have not found a case where such a construct would be needed. The exploration of this topic remains future work.

In Figure 4 the each fragment is used to express that after the subject has called its own notify method, it must call the update method of each of its observers.

4. REALIZATION

The RECLIPSE tool suite [21] has been implemented as a collection of plug-ins for FUJABA4ECLIPSE, which is an integration of FUJABA [4] into the ECLIPSE framework.

Structural patterns are specified as special graph grammar rules. They describe the object structure that constitutes a given pattern in abstract syntax. These graph grammar rules are then translated to Story Diagrams [3] from which code is generated. The generated code realizes search algorithms for every structural pattern. The software under analysis is parsed into an abstract syntax graph representation. The search algorithms try to match the patterns in the abstract syntax tree and create annotations when a matching object structure is found (cf. Section 2). The annotations mark objects that represent relevant roles of a given pattern, e.g. for the *Observer* pattern the subject and the observer objects. Details on the structural pattern detection process can be found in [10].

In order to analyze the runtime behavior of a software system it is executed and the method calls that occur during execution are traced. To reduce the amount of data that has to be analyzed, not the complete behavior of the software is traced but only the instance behavior of previously annotated classes and methods, i.e. of pattern candidates that were identified during structural analysis. The behavior of objects of other types and other method calls is omitted. All pattern candidates are traced individually and, depending on the concrete program execution, a number of traces is generated for each of them.

The behavioral analysis algorithm assesses if each candidate's traces conform to the corresponding behavioral pattern. A trace conforms to the behavioral pattern when its method calls all conform to the pattern and when the trace represents a complete pass through the pattern. In this case the trace is accepted. If the trace contains method calls that violate the behavioral pattern, it is rejected. If the trace does not contain prohibited method calls but does not contain all mandatory method calls that are specified in the pattern, the software system may not have been executed long enough to collect sufficient data. In this case the trace is neither rejected nor accepted. Technically the analysis is performed by mapping the behavioral patterns to finite automata and using the traces as input for the automata. The traces can be accepted, not accepted or rejected by an automaton. For further information on the analysis process we refer to [23]. The ratio between accepted, not accepted and rejected traces enables the reverse engineer to judge if a given pattern candidate really is a pattern implementation.

5. EVALUATION

The dynamic design pattern detection approach in RECLIPSE [21] has been extended by the concepts presented in Section 3.

Wendehals evaluated his approach by analyzing parts of the ECLIPSE IDE in the version 2.1 [22]. That particular software was chosen because Gamma and Beck documented some of the design patterns employed in the design of the software [5]. Wendehals found that *Observer* implementations that were documented by Gamma and Beck were detected by the static analysis but rejected in the dynamic analysis step because of the problems described in Section 2.2.

We repeated the analysis using our extension of the approach and found that the dynamic analysis now was able to confirm the *Observer* candidates discovered in the static analysis. We also were able to detect implementations of the *State*, *Strategy* and *Chain of Responsibility* patterns. It was possible to tell *State* and *Strategy* implementations apart even though their static structure is identical.

6. RELATED WORK

Several approaches exist that use a combination of static and dynamic analysis to detect design patterns.

Brown [2] uses static and dynamic analysis to detect four of the patterns described in [6] in Smalltalk source code. The source code is transformed into two different models, one describing the static structure and the other describing method calls between objects at runtime. However, due to this separation of models, it is not possible to combine the analysis techniques. Three of the chosen patterns (*Composite*, *Decorator*, *Template Method*) are detected by analyzing the static model while the *Chain of Responsibility* pattern is detected in the dynamic model. The pattern detection algorithms are implemented manually and hence are not easy to extend or maintain.

Guéhéneuc and Ziadi [7] propose to extract UML 2.0 dynamic models such as sequence diagrams and statecharts from Java source code and carry out high level analyses, like conformance checking and pattern detection, on these models. Further results have however not been published to date.

Similar to our approach, Heuzeroth, Holl and Löwe use a dynamic analysis in order to improve results from a static one [8]. They define the pattern structure as relations on the elements of an abstract syntax graph. The static analysis finds tuples that satisfy these relations which are then used for the dynamic analysis. The behavior of the patterns is specified with pre- and postconditions in PROLOG. The authors state that their pattern specifications tend to get lengthy and complicated which reduces maintainability. An extension of their specification language, SAND-PROLOG, makes pattern specifications easier at the expense of their expressiveness. Conditions like "a class may not have any methods" cannot be expressed in SAND-PROLOG.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented an extension of the pattern detection approach described by Wendehals [22]. We introduced a new element to the pattern specification language to be able to deal with arbitrarily large sets of objects in behavioral pattern specifications. The extension was implemented for the RECLIPSE tool suite and evaluated for several patterns. It is now possible to correctly detect patterns that caused problems in Wendehals' original approach [14].

The approach still leaves open questions for future research. We intend to analyze larger software projects and try to detect more patterns to get a feeling for the scalability and expressiveness of our approach. Furthermore it would be interesting to quantitatively analyze the precision and recall of the extended RECLIPSE tool suite and compare it to similar reverse engineering tools. In the future we want to build upon the current reverse engineering techniques and use reverse engineered behavioral models to carry out further analyses like conformance checking.

8. REFERENCES

[1] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien. Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together. In D. Richardson, M. Feather, and M. Goedicke, editors, *Proc. of ASE-2001: The* 16^{th} *IEEE Conference on Automated Software Engineering*, pages 166–173, Coronado, CA, USA, November 2001. IEEE Computer Society Press.

[2] K. Brown. Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk. Master's thesis, North Carolina State University, June 1996.

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

[4] Fujaba Tool Suite. University of Paderborn, Germany. `http://www.fujaba.de`, last visit: September 2009.

[5] E. Gamma and K. Beck. *Contributing to Eclipse - Principles, Patterns, and Plug-Ins.* The Eclipse Series. Addison-Wesley, Boston, MA, USA, 2004.

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

[7] Y.-G. Guéhéneuc and T. Ziadi. Automated Reverse-Engineering of UML v2.0 Dynamic Models. In S. Demeyer, K. Mens, R. Wuyts, and S. Ducasse, editors, *Proc. of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*, pages 1–5, Glasgow, Scotland, UK, July 2005. Springer-Verlag.

[8] D. Heuzeroth, T. Holl, and W. Löwe. Combining Static and Dynamic Analyses to Detect Interaction Patterns. In *Proc. of the* 6^{th} *International Conference*

on Integrated Design and Process Technology, pages 1–7, Pasadena, Ca, USA, June 2002.

[9] C. Krämer and L. 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)*, pages 208–215, Monterey, CA, USA, November 1996. IEEE Computer Society Press.

[10] J. Niere. *Incremental Design-Pattern Recognition.* PhD thesis, University of Paderborn, Paderborn, Germany, 2004. In German.

[11] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards Pattern-Based Design Recovery. In *Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, FL, USA*, pages 338–348. ACM Press, May 2002.

[12] J. Niere, L. Wendehals, and A. Zündorf. An Interactive and Scalable Approach to Design Pattern Recovery. Technical Report tr-ri-03-236, University of Paderborn, Paderborn, Germany, January 2003.

[13] OMG. *UML 2.0 Superstructure Specification, Revised Final Adopted Specification (ptc/04-10-02)*, October 2004.

[14] M. C. Platenius. Berücksichtigung von Objektmengen bei der dynamischen Entwurfsmustererkennung. Bachelor thesis, University of Paderborn, 2009. In German.

[15] N. Shi and R. A. Olsson. Reverse Engineering of Design Patterns from Java Source Code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 123–134, Washington, DC, USA, September 2006. IEEE Computer Society.

[16] J. M. Smith and D. Stotts. SPQR: Flexible Automated Design Pattern Extraction From Source Code. In *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE 03)*, pages 215–224, Montreal, Canada, October 2003. IEEE Computer Society Press.

[17] I. Sommerville. *Software Engineering.* Addison Wesley, 4th edition, May 1992.

[18] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design Pattern Detection Using Similarity Scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, November 2006.

[19] 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.

[20] L. Wendehals. Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams. In *Proc. of the 6th Workshop Software Reengineering (WSR), Bad Honnef, Germany, Softwaretechnik-Trends*, volume 24/2, pages 63–64, May 2004.

[21] L. Wendehals. Reclipse, 2007. `http://www.reclipse.org`, last visit: September 2009.

[22] L. Wendehals. *Struktur- und verhaltensbasierte Entwurfsmustererkennung.* PhD thesis, University of Paderborn, Paderborn, Germany, 2007. In German.

[23] L. Wendehals and A. Orso. Recognizing Behavioral Patterns at Runtime using Finite Automata. In *Proc. of the 4th ICSE 2006 Workshop on Dynamic Analysis (WODA), Shanghai, China*, pages 33–40. ACM Press, May 2006.

APPENDIX

A. BEHAVIORAL PATTERN CHAIN OF RESPONSIBILITY

The *Chain of Responsibility* design pattern is another pattern where set objects can be used. The pattern intent is described in [6] as follows:

> ”Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.” [6]

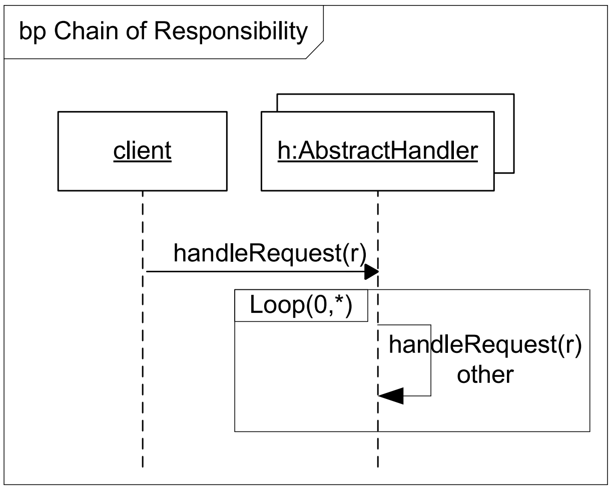


Figure 5: *Chain Of Responsibility* behavioral pattern

In Figure 5 there are two objects. `client` is an untyped object that delegates the request `r` to the handler chain. A set object represents an arbitrary number of `AbstractHandler` objects that constitute the chain. The `handleRequest` message represents the passing of the request along the chain.

Note the property `{other}` on the message, which is another language extension presented in [14]. A message from a set object to itself can either represent a method that is called by an object in the set on itself (i.e. the caller instance equals the callee instance) or a call of that method on another object in the set. To distinguish between these cases we introduced the keywords *self* and *other*.

The `other` call is enclosed by a loop fragment with the bounds 0 and *. Either the request is handled (and consumed) by the first handler in which case the loop fragment would be executed zero times or it is handled by an arbitrary handler somewhere in the chain.

Mapping Features to Domain Models in Fujaba

Thomas Buchmann
Angewandte Informatik 1
Universität Bayreuth
D-95440 Bayreuth
thomas.buchmann@uni-bayreuth.de

Alexander Dotor
Angewandte Informatik 1
Universität Bayreuth
D-95440 Bayreuth
alexander.dotor@uni-bayreuth.de

ABSTRACT
In the past, several approaches have been made to combine feature models and domain models on the level of class diagrams. But the model-driven development approach also covers models that describe the behavior of a software system. In this paper we present a mapping of feature models and configurations to executable model elements which is one step towards an overall model driven process for product line engineering. We present a tool called *MODPLFeaturePlugin*, which allows the user to establish a mapping between a distinct feature model and the domain model of the software system realized in Fujaba and also to create code for specific product configurations.

Keywords
version control, product lines, model-driven development, configuration management, code generation, feature mapping

1. INTRODUCTION
The term model-driven development [13] of software systems describes the creation of systems by specifying models instead of writing code. Usually these models are created in CASE tools which provide class diagrams to model the static structure of a software system. These kind of diagrams lack the ability to model variability explicitly. In the context of software product lines [5], feature models are used to model variability in a family of software systems.

Recently some approaches have been made to combine feature models and domain models created with CASE tools [1], [14], [10]. But model-driven development is more than just creating models that describe the static structure of a system - the behavior has to be described as well. The UML standard provides many behaviour modelling paradigms which are supported by some tools like *Rational Software Architect*[1] or *Topcased*[2]. Unfortunately not all of those behavioural models provide execution semantics or allow to generate executeable code. One of the key features of model-driven development compared to model-based development is the generation of executable code. Therefore we chose Fujaba for our current project called *MOD2-SCM*, in which we develop a model-driven product line for *Software configuration management* (SCM) systems [4]. The benefits of a model driven approach are (1) making the underlying models explicit, rather than having them implicitly defined in the program code, (2) providing reusable modules which can be combined in a flexible way through defining orthogonal components which are loosely coupled and (3) support rapid construction of new systems by providing a product line. This project is an example for the application of the general model-driven development process of software product lines, which is supported by *MODPLFeaturePlugin* (**mo**del **d**riven **p**roduct **l**ine feature mapping plugin for Fujaba).

[1]http://www.ibm.com/software/awdtools/swarchitect
[2]http://www.topcased.org

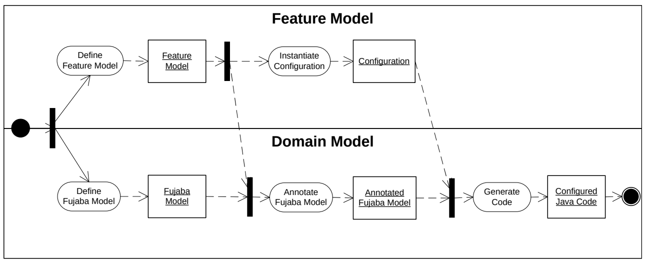


Figure 1: The MDD process which combines Feature models and domain models using annotations

The domain model of our product line consists of both class diagrams and behavioral diagrams developed in Fujaba. In the following we present *MODPLFeaturePlugin* which enables the Fujaba user to map elements from feature models to fujaba domain models and to generate code for specific product configurations. The plugin consists of two main parts: (1) an extension of the Fujaba editor to provide capabilities to establish a mapping between feature diagram elements and domain model elements and (2) a preprocessor for the Fujaba codegenerator, which is able to generate code for specific product configurations.

Even though the MODPLFeaturePlugin was developed within the MOD2-SCM project, it is by no means specific to the SCM domain and can be applied to all Fujaba models being composed of class diagrams and story diagrams.

2. MAPPING FEATURES TO FUJABA MODEL ELEMENTS
In our work we try to bridge the gap between features in a variability model and model elements of a system family. In a model-driven process a system configuration should be mapped automatically to the domain model, and code for the specific feature selection should be generated. In the context of product line engineering, feature models are widely used. The diagram in Figure 1 depicts our proposal

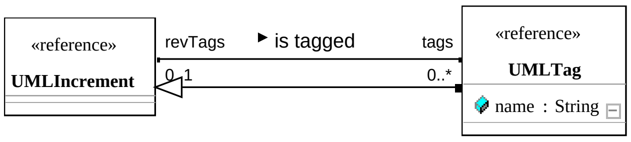


Figure 3: Excerpt from the Fujaba meta-model.

of a model-driven development process of software product lines, which is supported by *MODPLFeaturePlugin*. The domain model and the feature model are created in their respective tools, and a mapping between both models is established via *MODPLFeaturePlugin*. In the final step a configuration which is derived from the feature model is used to generate code for that specific configuration. Since there is no tool which supports the whole model-driven development process of software product lines, but several tools supporting single tasks in the whole development process (like *FeaturePlugin* which allows the creation of feature models, or UML tools which allow modeling the domain model), our work shows a way how to bridge the gap between feature modelling and domain modelling. The left part of Figure 2 shows the feature model of our model-driven pro-duct line of SCM systems. On the right hand side an excerpt of the domain model (particularly the package diagram and a class diagram) realizing those features is shown.

The feature model shown on the left side in Figure 2 was created using the tool *FeaturePlugin* [1]. It offers the option to export feature models and respective configurations via XML. *MODPLFeaturePlugin* is able to import feature models and configurations exported from *FeaturePlugin*.

Our approach establishes a mapping between feature model elements and their respective realizations in the domain model through attaching annotations containing the feature's unique identifier to the domain model elements. Fujaba's metamodel offers basic support for annotations by providing the class `UMLTag`. Instances of `UMLTag` can be attached to any element derived from `UMLIncrement` (see Figure 3).

A model element can also be decorated with multiple tags. The syntax that is used to describe the tag is similar to Java annotations: Each tag starts with the symbol "@" followed by a string literal which specifies the name of the tag and key-value pairs surrounded by parentheses (i.e. `@name(key=value)`) Examples are shown in Figure 4, like `@feature(id="directedDeltas")`. The annotation of model elements with the respective feature identifier can be performed on any level of granularity. On a coarse-grained level, units such as packages, classes and associations are decorated with features, whereas on a more fine-grained level, attributes, methods and even story patterns etc. can be decorated. The coarse-grained approach keeps the multivariant architecture manageable. But it is up to the modeler's discipline to use the feature annotations carefully.

In an extensive way of using feature annotations, the modeler may easily lose track and may face a degree of complexity which cannot be managed anymore. Therefore, several aids for the developer have been implemented in *MODPLFeaturePlugin*.

- First of all, the creation of tags, which annotate model elements, has been made much easier by providing a class `FeatureTag`, which is derived from `UMLTag`. It has a fixed name **feature** and it only support one key-value pair: the unique identifier of the feature model element. *MODPLFeaturePlugin* also provides a special dialog for creating and editing such `FeatureTag`s. The modeler can only choose between the identifiers of an already existing feature model. The advantage over manually specified feature identifiers is integrity of configurations based on existing feature models and the annotated model elements.

- To keep track which model elements are used to realize a certain features, *MODPLFeaturePlugin* highlights all FeatureTags that match a feature when clicking on that specific feature in the tree view (c.f. in Figure 4 the feature **Forward Delta** has been selected in the tree view). Selected features in the tree view are symbolized by `FeatureTag`s with green background in the Fujaba editor.

- Additionally, the plugin also checks for missing features in the domain model (depicted by yellow triangles in the feature model with an expression mark in the feature tree and the text "[unused feature]") and it provides several constraint checkers. E.g. highlighting domain elements with feature tags violating an exclusive-or constraint definded in the feature model. The elements displayed with a red background in Figure 4 show that kind of constraint violations.

Additionally, propagation rules for the tagged elements to their dependent elements can be applied, to give the modeler an overview which other model elements (attributs, roles, associations, subclasses etc.) are affected by adding a `FeatureTag` to a domain model element. The automatically added tags are highlighted with a blue background (not shown in Figure 4). These rules are also applied automatically during the code generation process, cf. section 3. Enumerating the propagation rules is not possible due to space restrictions. All rules and their application can be found in [2].

MODPLFeaturePlugin was primarily designed to work with the Fujaba4Eclipse-SwingUI branch, but it can also be used with the stand-alone version of Fujaba. Please note that Eclipse specific extensions, like the separate view showing the feature tree is not supported in the stand-alone version.

3. GENERATING CODE

After a domain model has been tagged with the feature annotations, code for a specific configuration can be generated. For that, a configuration created with *FeaturePlugin* is loaded by our plugin (also via XML import). A configuration is a distinct selection of features from the feature model which describes a concrete product. This configuration is used to control the preprocessor of the codegenerator.

The preprocessor is a plugin to the Fujaba Codegen2 plugin [8]. Since Codegen2 provides a so called "chain-of-responsibility" to address codewriters for different model elements or

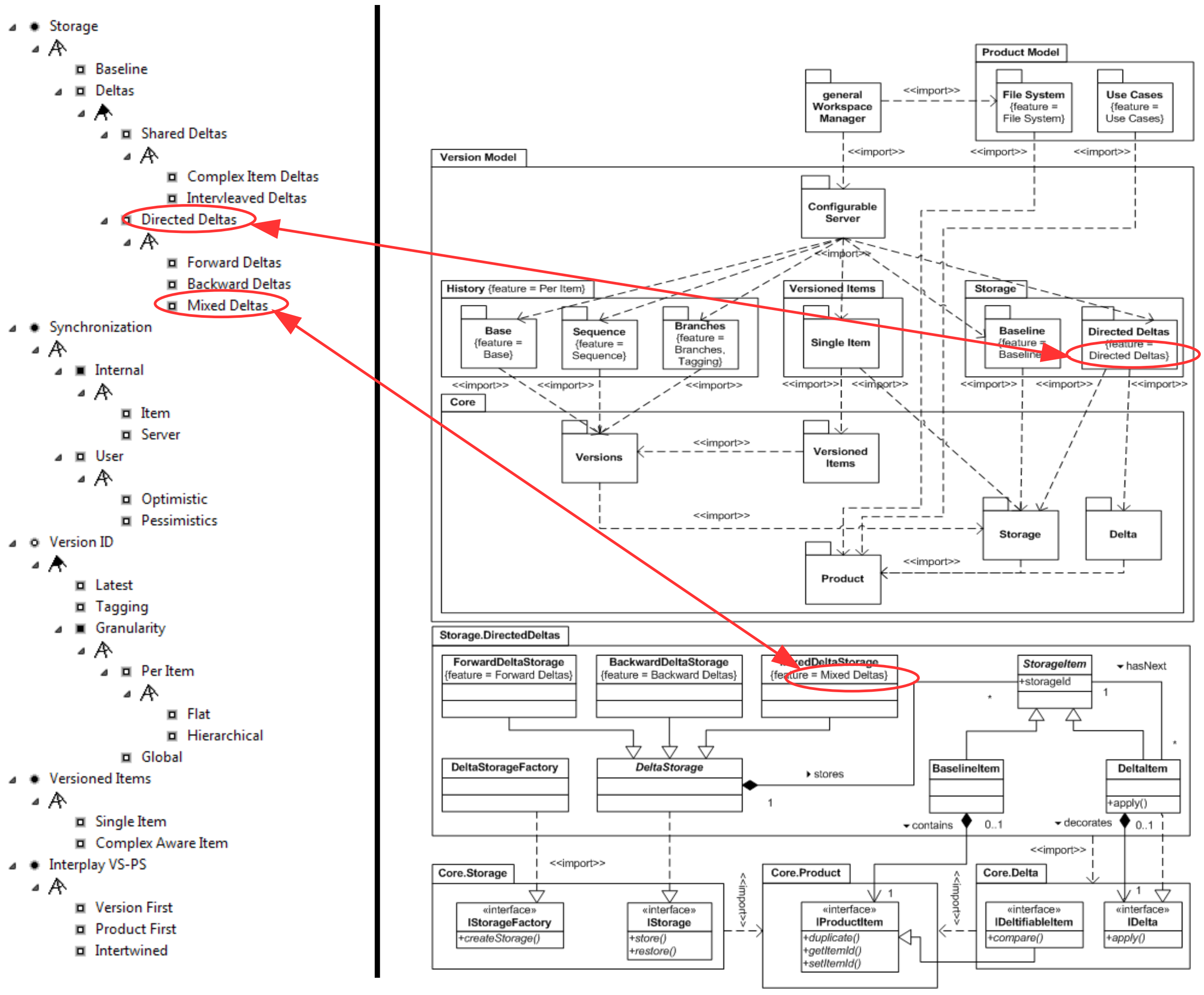
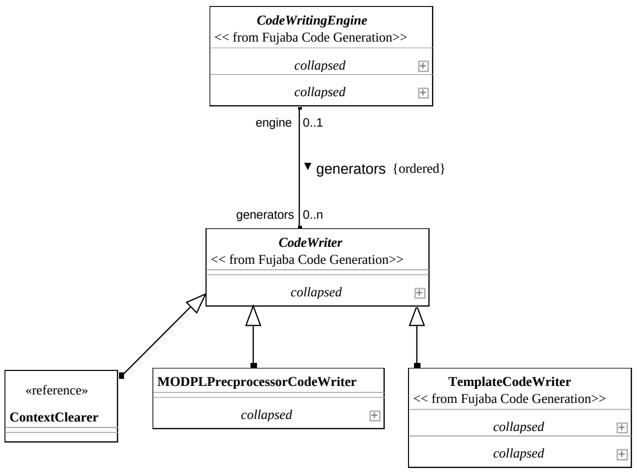


Figure 2: Conceptual dependency between features and model elements

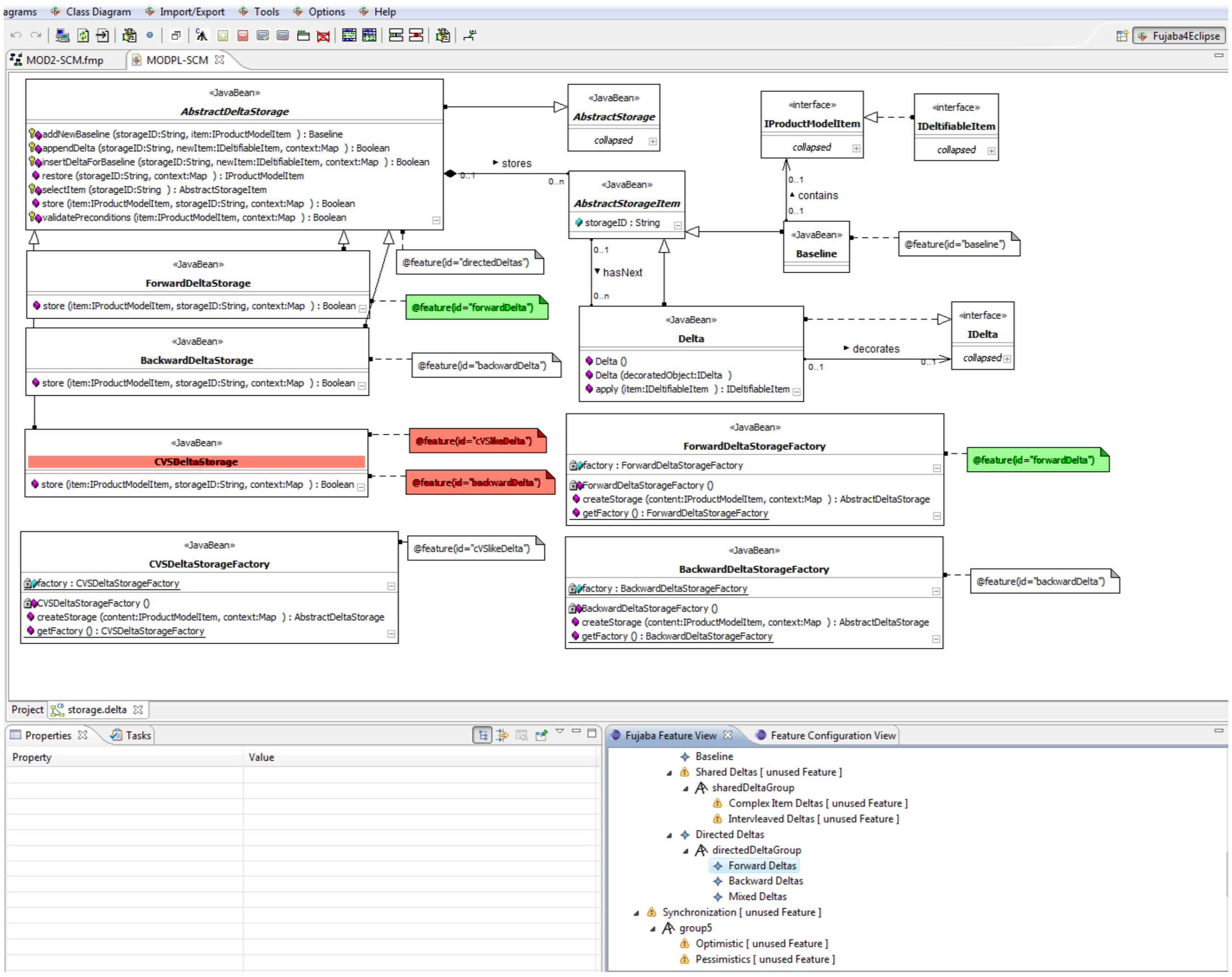
purposes (see Figure 5), it was fairly easy to add a preprocessor to this chain. During the code generation process, these feature annotations are evaluated against a given system configuration which is specified in FeaturePlugin [1]. If a feature is selected in a configuration the model elements with its name have to be part of the configured domain model. Each model element can be tagged by multiple features, in which case they are evaluated analogous to a logical *and* (i.e., all features have to be selected). This means also, that untagged model elements are always part of the configured model. For all model elements which contain tags representing features not part of the current configuration no code is generated. This process is very similar to the preprocessor step in compilers. As a consequence we face the same problems as compiler preprocessors: it is very easy to produce syntactically wrong code. Therefore we set up several consistency rules [2] that have to be met to ensure syntactical correctness of the resulting code.



CodeWritingEngine
<< from Fujaba Code Generation>>
collapsed
engine 0..1
generators {ordered}
generators 0..n
CodeWriter
<< from Fujaba Code Generation>>
collapsed
«reference» ContextClearer
MODPLPrecprocessorCodeWriter
TemplateCodeWriter
<< from Fujaba Code Generation>>

Figure 5: MODPLPreprocessorCodeWriter in the Codegen2 context.

4. RELATED WORK



Class Diagram
Import/Export
Tools
Options
Help
Fujaba4Eclipse
MOD2-SCM.fmp
MODPL-SCM
«JavaBean» AbstractDeltaStorage
addNewBaseline (storageID:String, item:IProductModelItem) : Baseline
appendDelta (storageID:String, newItem:IDeltifiableItem, context:Map) : Boolean
insertDeltaForBaseline (storageID:String, newItem:IDeltifiableItem, context:Map) : Boolean
restore (storageID:String, context:Map) : IProductModelItem
selectItem (storageID:String) : AbstractStorageItem
store (item:IProductModelItem, storageID:String, context:Map) : Boolean
validatePreconditions (item:IProductModelItem, context:Map) : Boolean
«JavaBean» AbstractStorage
collapsed
stores
«JavaBean» AbstractStorageItem
storageID : String
«interface» IProductModelItem
collapsed
«interface» IDeltifiableItem
collapsed
contains
«JavaBean» Baseline
@feature(id="baseline")
hasNext
«JavaBean» ForwardDeltaStorage
store (item:IProductModelItem, storageID:String, context:Map) : Boolean
@feature(id="directedDeltas")
@feature(id="forwardDelta")
«JavaBean» BackwardDeltaStorage
store (item:IProductModelItem, storageID:String, context:Map) : Boolean
@feature(id="backwardDelta")
«JavaBean» Delta
Delta ()
Delta (decoratedObject:IDelta)
apply (item:IDeltifiableItem) : IDeltifiableItem
decorates
«interface» IDelta
collapsed
«JavaBean» CVSDeltaStorage
store (item:IProductModelItem, storageID:String, context:Map) : Boolean
@feature(id="cVSlikeDelta")
@feature(id="backwardDelta")
«JavaBean» ForwardDeltaStorageFactory
factory : ForwardDeltaStorageFactory
ForwardDeltaStorageFactory ()
createStorage (content:IProductModelItem, context:Map) : AbstractDeltaStorage
getFactory () : ForwardDeltaStorageFactory
@feature(id="forwardDelta")
«JavaBean» CVSDeltaStorageFactory
factory : CVSDeltaStorageFactory
CVSDeltaStorageFactory ()
createStorage (content:IProductModelItem, context:Map) : AbstractDeltaStorage
getFactory () : CVSDeltaStorageFactory
@feature(id="cVSlikeDelta")
«JavaBean» BackwardDeltaStorageFactory
factory : BackwardDeltaStorageFactory
BackwardDeltaStorageFactory ()
createStorage (content:IProductModelItem, context:Map) : AbstractDeltaStorage
getFactory () : BackwardDeltaStorageFactory
@feature(id="backwardDelta")
Project
storage.delta
Properties
Tasks
Property
Value
Fujaba Feature View
Feature Configuration View
Baseline
Shared Deltas [unused Feature]
sharedDeltaGroup
Complex Item Deltas [unused Feature]
Interleaved Deltas [unused Feature]
Directed Deltas
directedDeltaGroup
Forward Deltas
Backward Deltas
Mixed Deltas
Synchronization [unused Feature]
group5
Optimistic [unused Feature]
Pessimistics [unused Feature]

Figure 4: Highlighting of selected features and constraint violations.

Czarnecki et al. describe in their work about Mapping Features to Models [6] a tool to establish a bidirectional mapping between feature models and Ecore elements, based on Ecore class diagrams (see [14]), called *Ecore.fmp*. It uses many of the same notations and display elements as a previous version named *FeaturePlugin* [1]. Unlike FeaturePlugin, which focuses strict-ly on feature modelling in an isolated context, Ecore.fmp aims to create Ecore compliant class diagrams out of existing feature models and vice-versa. In the current version of Ecore.fmp, the creation of a feature model from an existing Ecore model is not yet supported properly. The creation of Ecore model files out of feature models also still needs to be implemented. Since it is tightly coupled with Ecore, it does not support arbitrary EMF-models or even executable models. Furthermore a 1:1 mapping of a feature model to a class model is not appropriate in many cases: E.g. cross-cutting feature such as ”Synchronization” in our example shown in the feature model in Fig. 2, mapping features to model elements of different granularity such as classes or methods - not every feature may be realized by a class. But also UML class diagrams do not provide capabilities to model variability in terms of cardinality of association ends. The realization of a feature could require a to-one association, whereas the realization of another feature could require a to-many end of the same association. According to the project website, there is no further development of Ecore.fmp.

Heidenreich et al. developed a set of Eclipse plugins that also allows the user to establish a mapping between features and feature realisations (i.e., model elements) [11]. The underlying model (feature realisation) can be defined in arbitrary Ecore-based languages. It provides four different kinds of views, that visualize the current feature selection in different ways [9]. The plugin aims at supporting the developer in the complex task of defining mappings between features / configurations and their realizations. FeatureMapper provides no support for consistent model annotations. In MODPLFeaturePlugin, dependencies between model elements are taken into account which are specific to the meta-models for class diagrams and story diagrams. FeatureMapper does not know these dependencies, since it is a generic tool working with any meta-model defined in Ecore. FeatureMapper provides the possibility to configure models and to use that configured models in the further development process. Our approach allows to either configure the code directly (by using the java code generation engine) or to obtain a configured Ecore model, by using the EMF Code Generator [7]. The EMF-Adapter for Fujaba [12] could be a promising approach to combine FeatureMapper and Fujaba, but it is still in a very early stage. In case of a working EMF-Adapter, it could be feasible to try using FeatureMapper on Fujaba models.

There are also some commercial tools, that support modeling a product line by specifying feature models, like pure system's *pure::variants*[3]. These tools do not provide a model-driven process to develop a product line in a model-driven way. Usually they work on features provided on a file basis and only cover a small part of the product line process - variant management.

5. CONCLUSION

In this paper we presented *MODPLFeaturePlugin*, a plugin to Fujaba which aims at supporting a model-driven development process of software product lines in terms of realizing a mapping between elements from feature models and domain model elements. It provides several aids to the modeler and it supports the generation of specific product configurations based upon feature selection. Currently work is addressed to add several other constraint checkers to the editor plugin and to integrate it with the package diagram editor presented in [3].

6. ACKNOWLEDGEMENTS

We would like to thank our supervisor Prof. Dr. Bernhard Westfechtel for the invaluable input during the development of this plugin and his feedback as a reader of this paper. Further thanks go to (in alphabetical order): Leif Geiger, Ruben Jubeh and Dietrich Travkin for their hints how to extend Fujaba and the codegeneration engine.

7. REFERENCES

[1] M. Antkiewicz and K. Czarnecki. Featureplugin: Feature modeling plug-in for eclipse. In *OOPSLA '04 Eclipse Technology eXchange (ETX) Workshop*, Vancouver, British Columbia, Canada, Oct. 24-28 2004. ACM.

[2] T. Buchmann and A. Dotor. Constraints for a fine-grained mapping of feature models and executable domain models. In M. Mezini, D. Beuche, and A. Moreira, editors, *1st International Workshop on Model-Driven Product Line Engineering (MDPLE'09)*, volume 1 of *CTIT Workshop Proceedings*, pages 9–17. CTIT, 2009.

[3] T. Buchmann, A. Dotor, and M. Klinke. Supporting modeling-in-the-large in Fujaba. submitted for publication.

[4] T. Buchmann, A. Dotor, and B. Westfechtel. Model-driven development of software configuration management systems - a case study in model-driven engineering. accepted for publication.

[5] P. Clements and L. Northrop. *Software product lines: practices and patterns*, volume 0201703327. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[6] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE 05*, 2005.

[7] L. Geiger, T. Buchmann, and A. Dotor. EMF code generation with Fujaba. In L. Geiger, H. Giese, and A. Zündorf, editors, *Proceedings of the 5th International Fujaba Days*, 2007.

[8] L. Geiger, C. Schneider, and C. Reckord. Template- and modelbased code generation for MDA-Tools. In H. Giese and A. Zündorf, editors, *Proceedings of the 3rd International Fujaba Days.* Universität Paderborn, 2005.

[9] F. Heidenreich, I. Şavga, and C. Wende. On controlled visualisations in software product line engineering. In *Proceedings of the 2nd Int. Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008)*, Sept. 2008. To appear.

[10] F. Heidenreich, J. Kopcsek, and C. Wende. Featuremapper: Mapping features to models. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 943–944. ACM, May 2008.

[11] F. Heidenreich and C. Wende. Bridging the gap between features and models. In *2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07)*, 2007.

[12] J. Johannes. Letting EMF tools talk to Fujaba through adapters. In U. Assmann, J. Johannes, and A. Zündorf, editors, *Proceedings of the 6th International Fujaba Days 2008.* Technische Universität Dresden, Technische Universität Dresden, September 18-19 2008.

[13] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[14] M. Stephan and M. Antkiewicz. Ecore.fmp a tool for editing and instantiating class models as feature models. Technical report, University of Waterloo, 2008.

[3]http://www.pure-systems.com/

PropertyChange Events meet Fujaba Statecharts

Ruben Jubeh, Albert Zündorf
University of Kassel, Software Engineering,
Department of Computer Science and Electrical Engineering
Wilhelmshöher Allee 73
34121 Kassel, Germany
[ruben | zuendorf]@cs.uni-kassel.de
`http://www.se.eecs.uni-kassel.de/se/`

ABSTRACT
The current Fujaba Statechart semantics allows arbitrary event triggers, but currently Statechart Transitions can only be triggered by timers and by dedicated signal methods. While modeling an application, one still has the problem how and where to call these signal methods. This restricts especially reactive systems modeling, where event handling and dispatching are central key elements. In this proposal, we suggest to use PropertyChange-Events on arbitrary model objects as event triggers. This broadens the available event triggers to any change occurring in the model, without the need to implement the trigger manually.

1. INTRODUCTION
In this proposal we want to show how property change events can be used as statechart event triggers. Property change events are dispatched every time a runtime model change occurs, so this is a powerful event source. As running example, we will utilize a reactive heating control application which consists of a controller, a temperature sensor and a radiator. First we show how the problem may be modeled with the already existing statechart features of Fujaba, as described in [4] and [2]. Showing the weaknesses of the old approach, we will then propose a more direct handling of PropertyChange Events.

2. EXISTING APPROACH
Figure 1 shows the class diagram for the old-style solution of our application. To conform with common Java design standards, our Sensor has a corresponding SensorListener interface, which may be used to listen to sensor events. Our example has two states, idle and heating. Figure 2 shows the basic corresponding statechart.

In order to react to sensor change events, we need to register a listener at the sensor. The listener invokes the corresponding signal methods, which will fire the state transitions, finally, cf. Figure 3. Note that one needs to declare a listener implementing class, delegate the call to the corresponding signal method when the right conditions are met, and has to register the listener implementation on the sensor. This classic approach is error prone and excessive. In addition, the specification of the reactive behavior is scattered between the statechart and the PropertyChange handling method, making it hard to assess the overall behavior during maintenance. Anyhow, if the sensor temperature reading changes and the sensor informs all listeners of that change (not shown), this will eventually fire the state transitions.

3. NEW NOTATION PROPOSAL
We are basically interested in any change of the temperature property of the TemperatureSensor class. By using the *JavaBean* stereotype on that class, Fujaba generates:

1. a generic observer pattern implementation based on the Java PropertyChange mechanism with a to-n-Association to PropertyChangeListeners
2. modified setters for each property (attributes and roles) that fire a PropertyChangeEvent when the property changes

This mechanism is also the basis for a Fujaba persistency mechanism and is stable, as described in [3].

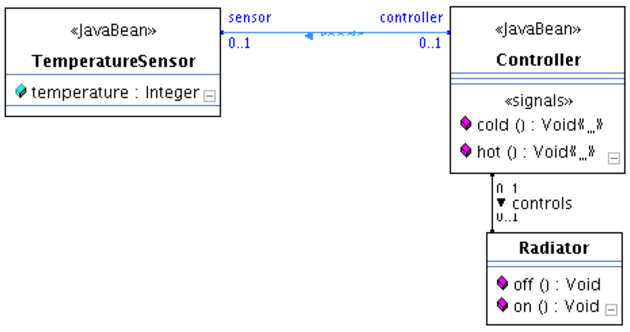


Figure 4: Revised Class Diagram with JavaBean stereotype added

We now use this as source of our event delegation chain. For the sake of simplicity, we will start with a statechart within the TemperatureSensor class itself. Figure 4 shows the updated class diagram with JavaBean stereotype added. Figure 5 shows a statechart example according to our new proposal. Instead of having event triggers corresponding to signal methods, we use the property temperature of the class TemperatureSensor as event trigger, directly. When detecting such a construct, the code generator now will generate code that:

1. adds a PropertyChangeListener for the property temperature of the instance itself

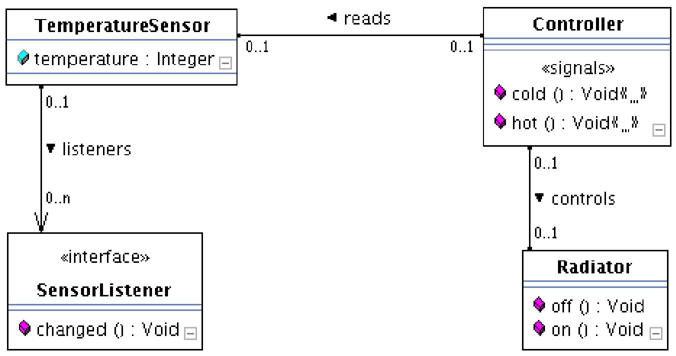


Figure 1: Example Domain Model

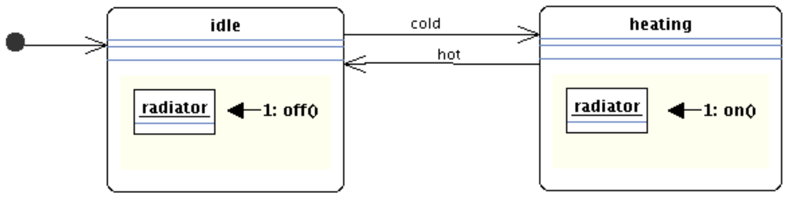
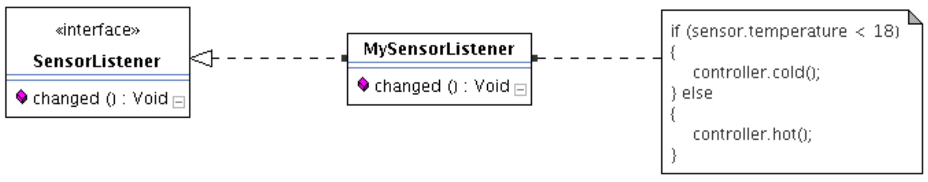


Figure 2: Heating Control Statechart



```
if (sensor.temperature < 18)
{
    controller.cold();
} else
{
    controller.hot();
}
```

Figure 3: Wrapping Change Events

2. dispatches a corresponding FEvent is to the statechart, when the listener is called

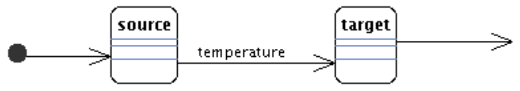


Figure 5: Property declared as Event Trigger

The semantics of such a state transition is as follows: the transition fires when the given property, in this case *temperature* of the this-instance, changes. Furthermore, one might want to add a condition to the transition by adding a boolean transition guard. When doing so, the code generator will now add the new property value as local variable to the guard method. So, compact conditions like *[temperature < 18]* are possible. Note that with this new approach, it is not necessary to deal with the SensorListener or, more generally, PropertyChangeListener, manually. The necessary implementation details are just in the generated code. Figure 6 shows an excerpt of the generated code within the statechart initialization method.

Back to the initial example, we want to have the statechart in the controller. Figure 7 shows the statechart according to our proposal. The temperature property is addressed indirectly by sensor.temperature. Note that is sufficient to specify just the guard expression. An expression parser will be used to extract the "sensor.temperature" string. The left side of that string, "sensor", will be recognized as role name (c.f. Figure 4) of the association reads between TemperatureSensor and Controller, so it resolves to an associated TemperatureSensor instance. The right side, will be resolved to the property of that instance. When detecting such a construct, the code generator will now generate code that does the following:

- It adds a PropertyChangeListener for the sensor property of the instance itself
- When the listener is called, it adds/removes a ProperyChangeListener to the TemperatureSensor instance
- When the latter listener is called, a corresponding FEvent is dispatched to the statechart

So, we use nested PropertyChangeListeners to allow properties of non-this objects as event trigger source. Note that the given example utilizes transition guards, for which the code generator will generate local variables as parameter of the guard method. Figure 8 shows an excerpt of the generated code for the given example within the statechart initialization method.

4. OPEN ISSUES AND FUTURE WORK

The given proposal works fine on the given running example, but for more compex systems, some aspects still need to be discussed:

1. For pure reactive systems, we want to fire transitions not only when the property changes, but also instantly when the guard condition is met. This means, whenever a state is entered, the guards of outgoing transitions that refer to model properties shall be evaluated and if a guard holds, the transition should fire. This semantic probably needs to be annotated.

2. It might be useful to have any property change of an object as trigger, for example *sensor.**. That would change the generated code minimally, but guard methods then need all properties of the resolved objects as local variables.

3. We want to support even more complex path expressions as e.g. *house.sensor[1].value* || *house.sensor[3].value*. This eventually leads to the necessity of an action language for transition guards.

4. Name conflicts for transitions can occur. It is possible to have a signal method with the same name as a property. At least the code generator should give a warning when detecting such a construct.

5. When the event triggering property resolves to a to-n association end, conditions might want to read additional properties of the CollectionChangeEvent. This needs to be addressed.

5. CONCLUSIONS

So far, we have elaborated the new concepts for an elegant handling of PropertyChange Events with Fujaba Statecharts and we have evaluated various examples to show the value of this new feature. These examples show that our proposal is efficient and it reduces the model complexity for a reactive system, significantly. We have started to implement this new approach through an adaption of the corresponding CodeGen2 templates (See [1] for details). An extension of Fujaba's meta model for Statecharts seems not necessary. Thus, we will be able to present a working code generation and running examples at the FujabaDays 2009.

This paper enables the use of Fujaba Statecharts for modeling reactions to model changes. Beyond this, we currently investigate the use of Fujaba Statecharts for modeling asynchronous remote procedure calls, RPCs, in case of Google Web Toolkit (GWT), where one needs to provide RPC result callback handlers. We want to model such callbacks with Fujaba Statecharts elegantly as well.

6. REFERENCES

[1] L. Geiger, C. Schneider, and C. Record. Template- and modelbased code generation for MDA-tools. In *3rd International Fujaba Days*, Paderborn, Germany, 2005.

[2] H. J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating uml diagrams for production control systems. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 241–251, New York, NY, USA, 2000. ACM.

[3] C. Schneider. *CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Spe rrkonzepten.* PhD thesis, 2007.

[4] A. Zündorf. Rigorous object oriented software development. Habilitation Thesis, University of Paderborn, 2001.

```
myFReactive.makeTransition (source, target, "temperature", null, null);
this.addPropertyChangeListener("temperature", new PropertyChangeListener() {

   public void propertyChange(PropertyChangeEvent evt)
   {
      FEvent event = new FEvent(evt.getPropertyName());
      event.addToArgs(int.class, evt.getNewValue());
      myFReactive.enqueueEvent(event, FReactive.EXTERNAL_EVENT);
   }

});
```

Figure 6: Generated code for an property change event trigger

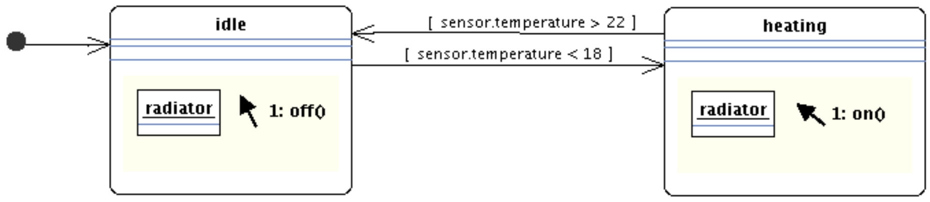


Figure 7: Indirect Property declared as event trigger with condition

```
myFReactive.makeTransition (idle, heating, "sensor.temperature",
      "guard8264334ForSensor.temperatureFromIdleToHeating", null);
this.addPropertyChangeListener("sensor", new PropertyChangeListener()
{
   PropertyChangeListener inner = new PropertyChangeListener()
   {
      public void propertyChange(PropertyChangeEvent evt)
      {
         FEvent event = new FEvent(evt.getPropertyName());
         event.addToArgs(int.class, evt.getNewValue());
         myFReactive.enqueueEvent(event, FReactive.EXTERNAL_EVENT);
      }
   };
   public void propertyChange(PropertyChangeEvent evt)
   {
      final TemperatureSensor oldValue = (TemperatureSensor) evt.getOldValue();
      if (oldValue != null)
      {
         oldValue.removePropertyChangeListener("temperature", inner);
      }
      final TemperatureSensor newValue = (TemperatureSensor) evt.getNewValue();
      if (newValue != null)
      {
         newValue.addPropertyChangeListener("temperature", inner);
      }
   }
});
```

Figure 8: Generated code for a nested/indirect property change event trigger

Evolution of Modelling Languages

Bart Meyers
Modelling, Simulation and Design Lab (MSDL)
University of Antwerp
Middelheimlaan 1
B-2020 Antwerp, Belgium
Bart.Meyers@ua.ac.be

Hans Vangheluwe
Modelling, Simulation and Design Lab (MSDL)
University of Antwerp
Middelheimlaan 1
B-2020 Antwerp, Belgium
Hans.Vangheluwe@ua.ac.be

ABSTRACT
Over the course of the complete life cycle of complex software-intensive systems and more importantly of entire product families, evolution is inevitable. Not only instance models, but also entire modelling languages are subject to change. This is in particular true for domain-specific languages. Up to this day, modelling languages are evolved manually, with tedious and error-prone migration of for example instance models as result. This position paper discusses the different evolution scenarios for various kinds of modelling artifacts, such as instance models, meta-models and transformation models. Subsequently, evolution is de-composed into four primitive scenarios such that all possible evolutions can be covered. The pre-requisites for implementing this approach are discussed, showing how a number of these are not yet supported by Fujaba. We suggest that using our structured approach in Fujaba will allow a relatively straightforward implementation of (semi-)automatic model evolution.

1. INTRODUCTION
In software engineering, the evolution of software artifacts is ubiquitous. These artifacts can be programs, data, requirements, documentation, but also languages. Language evolution applies in particular to domain-specific modelling (DSM), where domain-specific languages (DSLs) are specifically designed to minimize accidental complexity by using constructs closely coupled with their domain. This results in a reported productivity increase of a factor 5 to 10 [10]. DSLs must be quickly built and used, and grow incrementally. A formal underpinning for DSM is given by multi-paradigm modelling (MPM) [15].

The high dependence on their domains and the need for instant deployment make DSLs highly susceptible to change. Such an evolution of a language can have substantial consequences, which will be explained throughout this paper. Early adopters of the model-driven engineering paradigm dealth with this evolution problem manually. However, such a pragmatic approach is tedious and error-prone. Without proper methods, techniques and tools to support evolution, model-driven engineering in general and domain-specific modelling specifically will not scale to industrial use.

$$\begin{tikzcd}
 & MM & & MM_{T_i} \\
MM_{\kappa_i} & m \arrow[u, dashed] \arrow[rr, "T_i"] \arrow[dl, leftrightarrow, "\kappa_i"'] \arrow[ddr, "M"] & & T_i(m) \arrow[u, dashed] \\
\kappa_i(m) \arrow[u, dashed] & & MM_M & \\
 & & M(m) \arrow[u, dashed] &
\end{tikzcd}$$

Figure 1: A model and its relations in MPM.

1.1 Modelling Languages
To allow for a precise discussion of language evolution, we briefly introduce the concepts fundamental to modelling languages, in the context of multi-paradigm modelling [5].

The two main aspects of a model are its *syntax* (how it is represented) and its *semantics* (what it means).

Firstly, the syntax comprises *concrete syntax* and *abstract syntax*. The concrete syntax describes how the model is represented (in 2D vector graphical form for example), which can be used for model input as well as visualization. The abstract syntax contains the essence of the structure of the model (as an abstract syntax graph), which can be used as a basis for semantic anchoring. A single abstract syntax may be represented by multiple concrete syntaxes. There exists a mapping between a concrete syntax and its abstract syntax, called the *parsing mapping function*. There is also an inverse mapping, called the *pretty printing mapping function*. Mappings are usually implemented, or can be at least represented, as model transformations.

Secondly, the semantics of a model are defined by a complete, total and unique *semantic mapping function* which maps every model in a language onto an element in a *semantic domain*, such as differential equations, Petri Nets, or the set of all behaviour traces. Semantic mapping functions are performed on the abstract syntax for convenience.

A meta-model is the finite and explicit description of the abstract syntax of a language. Often, the concrete syntax is also described by (another) meta-model. Semantics are however not covered by the meta-model. The abstract syntax of the semantic domain itself will of course conform to a meta-model in its own right.

Figure 1 shows the different kinds of relations a model *m* is involved in. Relations are visualized by arrows, "conform to"-relationships are dotted arrows. The abstract syntax model *m* conforms to its meta-model *MM*. There is a bidirectional relationship κ_i (parsing mapping function and pretty printing mapping function) between *m* and a concrete syntax $\kappa_i(m)$. $\kappa_i(m)$ conforms to its meta-model MM_{κ_i}. Semantics are described by the semantic mapping function *M*, and map *m* to a model *M(m)*. *M(m)* has syntax which conforms to MM_M. Additionally, there may be other transformations T_i defined for *m*.

2. RELATED WORK

In this section other work related to evolution is presented and some useful concepts are introduced.

2.1 Model Differencing

In order to be able to model evolution in-the-large, one should be able to model differences between two versions of a model. This can of course be done by using lexical differencing, as used for text files, on the data representation of the model. However, the result of such analysis is often not useful, as (1) the differences occur at the granularity level of nodes, links, labels and attributes and (2) models are usually not sequential in nature and equivalences between models will not be taken into account. Hence, model differencing should be done at the appropriate level of abstraction. Some useful research has been done in this area [1, 17, 13, 23, 4]. Existing approaches typically rely on the abstract syntax graphs (ASGs) of the two models, and mainly traverse both graphs in parallel. Nodes in the graphs are matched by matching unique identifiers [1, 17], or by a number of heuristics [13, 23]. However, no large-scale version control system that computes the differences between graph-like models exists yet.

Next to the problem of finding differences, one should be able to *represent* them as a model, which we will call the *delta model*. There are two kinds of representations: operational and structural representations. In the operational representation, the difference between two versions of a model is modelled as the edit operations (create/read/update/delete) that were performed on the on one model to arrive at the other [1, 8]. When these operations are recorded live from a tool, this strategy is quite easy and powerful, but dependent on that particular tool and hard to visualize. In structural representations, either the model (or its DOM representation) is coloured [17, 23, 13, 19] or a designated delta model is created which can be used by modelling tools as yet another model [4, 20].

2.2 Model Co-Evolution

When the syntax of a modelling language evolves (i.e., the meta-model evolves), the most obvious side-effect is that its instance models are not conform to the new meta-model. Therefore, the co-evolution of models has become a popular research topic. This research is inspired by evolution in other domains, such as grammar evolution [18], database schema evolution [2] and format evolution [12].

It is widely accepted that a model co-evolution (i.e., migration) is best modelled as a model transformation [24, 11, 20, 9, 7, 22, 21, 3, 8], which we will call the *migration transformation*. Grushko et al. write this transformation manually using the Epsilon Transformation Language (ETL) [7].

Most of the approaches however define some specific operations as building blocks for evolution, similar to the operational representation of model differences. Such operations typically include "create meta-class", "restrict multiplicity on meta-association" or "rename meta-attribute" and are related to object-oriented refactoring patterns. These operations, which we will call *delta operations*, are reusable. Conveniently, migration transformations can be generated from sequences of delta operations. It is important that any possible evolution can be modelled, but there is a general consensus that the proposed sets of delta operations do not suffice. In a very recent approach, Herrmannsdörfer et al. try to solve this problem by repeatedly extending their list of delta operations [8]. In addition, they support customized evolution. This ensures expressiveness, but the migration transformation code must be implemented manually.

Table 1: Evolution operations as presented in [3].

Operation type	Operation
Non-breaking operations	Generalize meta-property Add (non-obligatory) meta-class Add (non-obligatory) meta-property
Breaking and resolvable operations	Extract (abstract) superclass Eliminate meta-class Eliminate meta-property Push meta-property Flatten hierarchy Rename meta-element Move meta-property Extract/inline meta-class
Breaking and unresolvable operations	Add obligatory metaclass Add obligatory metaproperty Pull metaproperty Restrict metaproperty Extract (non-abstract) superclass

Gruschko et al. make a distinction between non-breaking, resolvable and unresolvable operations. Non-breaking operations do not require co-evolution. Inconsistencies caused by resolvable operations can be resolved by co-evolution. However, model co-evolution for unresolvable operations requires additional information in order to execute. For example, when a "create obligated meta-feature"-operation is performed on a meta-model, then a new feature is created for each instance. However, the information about what the initial value of this feature will be, is unknown, as it differs from model instance to model instance. As an illustration, the operations proposed by Cicchetti et al. [3] are shown in Table 1. Note the similarities with refactoring patterns.

3. EVOLUTION FOR MPM

While model co-evolution as described above implements automation to some extent, there are other artifacts that might have to co-evolve. This section presents an exhaustive survey of possible evolutions and co-evolutions.

3.1 Syntactic Evolution

To get a general idea of the consequences of evolution, let us go back to Figure 1. When *MM* evolves, all models *m* have to co-evolve, which was discussed in Section 2.2. However, as the relations of Figure 1 suggest, the evolution of *MM* might affect other artifacts. First, similar to *m*, (the domain and/or image of) transformations such as κ_i, T_i and *M* might no longer conform to the new version of the metamodel. As a consequence, they too have to co-evolve. This makes all relations (syntactically) valid once again, which means that the system is syntactically *consistent* again. In short, meta-model evolutions can only be useful when both their model instances and related transformation models can co-evolve.

However, there are more scenarios. Firstly, it is possible that the meta-model changes in such a way that the co-evolved models become structurally different, for example by removing a language construct. This means that each transformation defined for each co-evolved model has to be re-executed. The resulting co-evolved models can also be structurally different, so a chain of required evolution transformation executions may be required.

Secondly, changes made to one meta-model can reflect on another meta-model. For example, when a meta-element is added to a meta-model, a new meta-element is often also added to the meta-model of the concrete syntax(es) in order to be able to visualize this new construct. A similar effect can occur between any two related (by transformation) meta-models. In this sense, a chain of meta-model changes is again possible.

Thirdly, until now, we only discussed meta-model evolution as the driving force. Evolution of other artifacts, such as instance models and transformation models should also be taken into account. The case of the evolution of a model is trivial: related models can co-evolve by executing the respective transformations. Note however that a co-evolved model may be a meta-model, so that might trigger a number of co-evolutions of its own.

The case of the evolution of a transformation model can get complicated. In many cases though, the evolved transformation simply has to be executed again on each model it is defined for. However, this would restrict a transformation evolution to remain compliant to its source and target metamodels, which is not always what we want. For example, it might be possible that a new language is created by mapping rules for each language construct of an existing language. This is in particular convenient for creating a concrete syntax. On top of this, there are two additional special cases of transformation evolution. Firstly, the evolution of the parsing mapping function or the pretty printing mapping function requires the other one to co-evolve in order to maintain a meaningful relation between abstract and concrete syntax. Such a co-evolution can be generalized to any bidirectional transformation. Secondly, the evolution of the semantic mapping function requires a means to reason about semantics in order to trigger co-evolution, which brings us to the concept of semantic evolution.

3.2 Semantic Evolution

As mentioned above, semantics of a model are defined by its semantic mapping function to a semantic domain. Some analysis can be performed on models in this semantic domain (for example: check for a deadlock in a Petri Net). The results of this analysis can be considered a *property* of the model, or *P(m)*. A semantic mapping function is constructed in such a way that some properties $P_M(m)$ hold both for a model and for its image under the semantic mapping (i.e., the intersection of both property sets). These common properties have to be maintained throughout evolution. An evolution is a semantic evolution if some of these properties change. This typically happens when the requirements of a system change.

In general, when a model *m* in a formalism whose semantics is given by semantic mapping function *M* evolves to *m'*, then $P_M(m')$ must be exactly $P_M(M(m))$ modulo the intended semantic changes. In general, when two versions of a system are (a) equal modulo their intended syntactic and semantic changes and (b) syntactically consistent, then the evolution of the system is *continuous*. Only continuous evolutions are deemed correct (and meaningful).

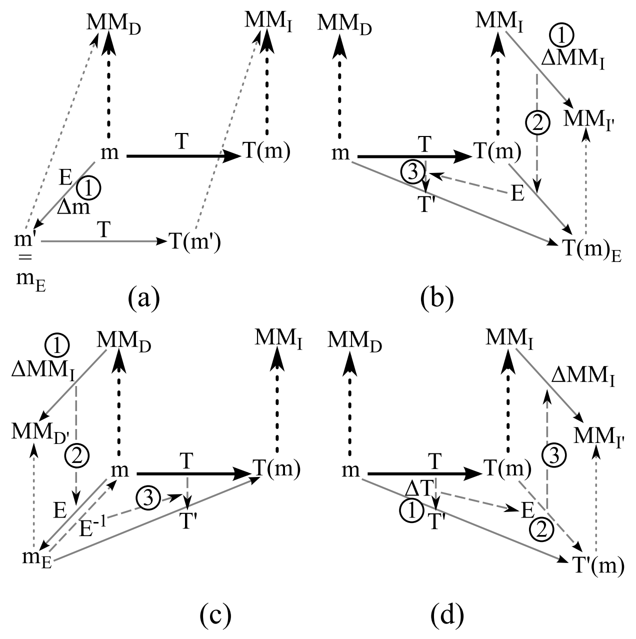


Figure 2: Co-evolution in (a) model evolution, (b) image evolution, (c) domain evolution and (d) transformation evolution.

4. DE-CONSTRUCTING EVOLUTION

As discussed in the previous section, there are infinitely many possible co-evolution scenarios. Nevertheless, these scenarios can always be broken down into a few basic ones. Figure 2 shows the possibilities. Again, arrows are transformations and dotted arrows are "conforms to"-relationships. Dashed arrows denote a (semi-)automatic generation. Each diagram starts from a bold relation between two meta-models MM_D and MM_I, modelled as a transformation *T* of models *m*.

4.1 Model Evolution

Figure 2 (a) shows model evolution. Some model *m* evolves to *m'*. In step 1 (the only step), a delta model $\Delta m$is constructed (either automatically or manually) that models the evolution of *m* to *m'*. This means that $m' = m + \Delta m$. The evolution itself is typically represented as a migration transformation, namely *E*. The equation $m_E = m + \Delta m = m'$ is valid. As previously discussed, because *m* evolved to *m'*, every transformation *T* must be executed again, resulting in *T(m')*, conform to MM_i.

4.2 Image Evolution

Image evolution is shown in Figure 2 (b). Suppose that a meta-model MM_I evolves to $MM_{I'}$. In step 1 a delta model ΔMM_I is constructed to represent the difference between MM_I and $MM_{I'}$. In step 2 a migration transformation *E* is generated out of ΔMM_I. The execution of *E* co-evolves models $T(m)$ to $T(m)_E$, so that they conform to the new meta-model $MM_{I'}$. Moreover, the execution transformation *T* has to result in valid models (i.e., conform to $MM_{I'}$). As a consequence, *T* has to co-evolve to a new transformation *T'* (as in step 3), which is able to transform every possible *m* that conforms to MM_D, to $T(m)_E$. The diagram presents a solution for the

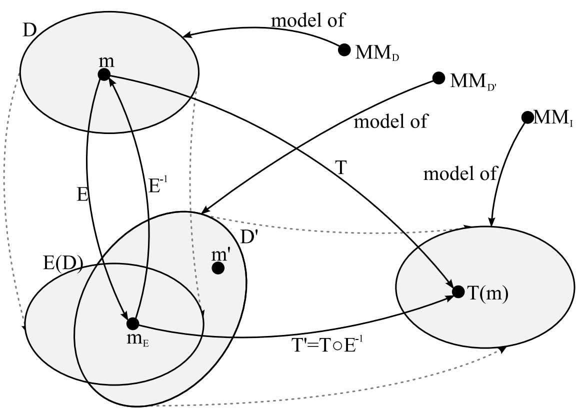


Figure 3: Set representation of domain-evolution. The evolution *E(D)* does not map onto *D'* exactly. For *m'*, the constraint *T'* = *T*∘*E*$^{-1}$ does not hold!

generation of this *T'*: for every *m*, *T'(m) = E(T(m))* holds, or in short, *T'* = *E* ∘ *T*. The co-evolution *T'* can be simply composed out of *T* and *E*.

4.3 Domain Evolution

Figure 2 (c) shows domain evolution, where MM_D evolves. The artifacts that co-evolve are similar to image evolution. This time however, *T* can be expressed as $T' = T \circ E^{-1}$. So, in this case, an inverse transformation E^{-1} needs to be constructed. Unfortunately, this equation does not hold for the entire domain *D'*, as shown in Figure 3. The migration transformation *E* projects the entire domain *D* to *E(D)*, but it is possible that $E(D) \neq D'$. For *m* in Figure 3 it may be possible possible to construct E^{-1} such that $T'(m_E) = T(E^{-1}(m_E))$ holds. However, for *m'*, which is an element of $D' \setminus E(D)$, this is not possible. Nevertheless, *T'* must apply to its entire domain *D'*, so the equation $T' = T \circ E^{-1}$ can not be used for all possible models conform to $MM_{D'}$.

4.4 Transformation Evolution

Figure 2 (d) shows transformation evolution. The requirements of a system can change, resulting in the adjustment of the (desired) properties of a model. If transformations evolve according to a delta model ΔT, it is possible that they only have to be executed once again. In this case, the changes on the transformation are limited: the image of *T'* must conform to MM_i. As previously discussed, other artifacts might possible co-evolve. In this case, a migration transformation *E* must be composed from which a delta model ΔMM_i can be constructed.

4.5 Evolution Scenario Amalgamation

Using a combination of these four scenarios, all possible evolutions can be carried out. Note however that the problem of Figure 3 applies, so automated co-evolution is not always possible. The so-called unresolvable changes can be classified as models in $E(D) \setminus D'$. On the other hand, the transformation has to support the models in $D' \setminus E(D)$. We call this the *projection problem*. In general, the projection problem arises when $dom_E(T) \not\subseteq dom(T)$.

5. PRE-REQUISITES FOR EVOLUTION

Following the discussion above, the proposed approach depends on a few more general techniques. Many of these pre-requisites are currently not supported by Fujaba. As a consequence, implementing all forms of evolution is currently not feasible in practice in Fujaba. The following pre-requisites (in order of priority) are necessary or at least useful for implementing evolution:

- *higher order transformation*: the automatic generation of migration transformations out of delta models requires support for higher order transformations, which are transformations that take other transformations as input and/or output. This is not supported by Fujaba, as the transformation language is not modelled explicitly (i.e., the meta-model is not available). There are several other uses for higher order transformation, in and out of the context of evolution, which are not discussed here [3, 16, 14], making higher order transformation a valuable feature in any MDE-tool;

- *model differencing*: in order to support automated evolution on a industrial level, it must be possible to generate delta models out of two versions of a model. Moreover, it is desirable that the activity of meta-modelling does not have to change in order to support automated evolution. The Fujaba Difference Tool Suite uses the SiDiff framework to calculate and visualize the difference between two models' XMI documents [19]. A so-called Difference Viewer Plugin shows a coloured difference model in Fujaba;

- *transformation inversing*: in order to automatically co-evolve a transformation in domain evolution, the inverse of the migration transformation is needed. In Fujaba this is implicitly featured by providing the possibility to implement bidirectional transformations using Triple Graph Grammars (TGGs) in MoTE [6]. However, in that case, one is restricted to the use of bidirectional transformation with triple graph grammars. It remains an open question whether TGGs are expressive enough to obtain the inverse of the migration transformation (which may for example delete elements).

- *representation of semantics*: as not only the syntax but also semantics of a modelling language evolves, there must be a way to represent these semantic changes. A more precise means to reason about semantics preservation (through properties?) is needed.

If all of these pre-requisites are implemented in Fujaba, a framework for evolution can be relatively easily implemented.

6. CONCLUSIONS

Extensive adoption of model-driven engineering is obstructed by the lack of support for automated evolution. Especially in domain-specific modelling, modelling languages are used while under development or under ceaseless change. When such languages evolve, support for (semi-)automated co-evolution must be available. To this day, research has been done only to support model co-evolution for meta-model evolution. Transformations or semantics are not yet taken into account.

We addressed this problem by de-constructing all possible (co-)evolution processes into four basic scenarios, which can be combined. We showed that the co-evolution of transformations can be problematic, because a transformation always needs to be able to transform all possible elements in its domain.

We discussed the pre-requisites for an implementation of automated evolution. It turns out that, in order to implement support for evolution, a number of pre-requisites, such as higher order transformation, model differencing, transformation inversing and semantics represention, have to be dealt with. Like all other current tools, Fujaba only supports a few of these, with higher order transformation as major absentee.

7. ACKNOWLEDGMENTS

We would like to thank Tihamér Levendovszky for the fruitful discussions on the subject of this paper.

8. REFERENCES

[1] M. Alanen and I. Porres. Difference and union of models, 2003.

[2] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Rec.*, 16(3):311–322, 1987.

[3] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *EDOC '08: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society.

[4] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[5] H. Giese, T. Levendovszky, and H. Vangheluwe. Summary of the workshop on multi-paradigm modeling: Concepts and tools. In T. Kühne, editor, *Models in Software Engineering Workshops and Symposia at MoDELS 2006*, volume 4364 of *LNCS*, pages 252–262. Springer-Verlag, October 2006.

[6] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy*, volume 4199 of *Lecture Notes in Computer Science (LNCS)*, pages 543–557. Springer Verlag, 10 2006.

[7] B. Gruschko, D. Kolovos, and R. Paige. Towards synchronizing models with evolving metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution at IEEE European Conference on Software Maintenance and Reengineering (ECSMR)*, 2007.

[8] M. Herrmannsdoerfer, S. Benz, and E. Juergens. Cope - automating coupled evolution of metamodels and models. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 52–76, 2009.

[9] J. Hoessler, J. Soden, Michael, and H. Eichler. Coevolution of models, metamodels and transformations. *Models and Human Reasoning*, pages 129–154, 2005.

[10] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, March 2008.

[11] R. Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, Nov. 2004.

[12] R. Lämmel and W. Lohmann. Format Evolution. In *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.

[13] Y. Lin, J. Gray, and F. Jouault. Dsmdiff: A differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4, Special Issue on Model-Driven Systems Development):349–361, 2007.

[14] B. Meyers and P. Van Gorp. Towards a hybrid transformation language: Implicit and explicit rule scheduling in story diagrams. Sixth International Fujaba Days, September 18–19 2008.

[15] P. J. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling: An introduction. In *SIMULATION80*, volume 9, pages 433–450, 2004.

[16] O. Muliawan. Extending a model transformation language using higher order transformations. *Reverse Engineering, Working Conference on*, 0:315–318, 2008.

[17] D. Ohst, M. Welle, and U. Kelter. Differences between versions of uml diagrams. *SIGSOFT Softw. Eng. Notes*, 28(5):227–236, 2003.

[18] M. Pizka and E. Jurgens. Automating language evolution. In *TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 305–315, Washington, DC, USA, 2007. IEEE Computer Society.

[19] M. Schmidt and T. Gloetzner. Constructing difference tools for models using the sidiff framework. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 947–948, New York, NY, USA, 2008. ACM.

[20] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15, April 2004.

[21] S. Vermolen and E. Visser. Heterogeneous coupled evolution of software languages. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 630–644, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] G. Wachsmuth. Metamodel adaptation and model co-adaptation. In E. Ernst, editor, *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer-Verlag, July 2007.

[23] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[24] J. Zhang and J. Gray. A generative approach to model interpreter evolution. In *OOPSLA Workshop on Domain-Specific Modeling*, pages 121–129, 11 2004. Vancouver, VC.

Fujaba hits the Wall(-e)

Pieter Van Gorp
Eindhoven University of Technology,
School of Industrial Engineering
De Lismortel 2
5600MB, Eindhoven, The Netherlands
p.m.e.v.gorp@tue.nl

Ruben Jubeh,
Bernhard Grusie
Kassel University,
Software Engineering Group
Wilhelmshöher Allee 73
34121 Kassel, Germany
ruben@cs.uni-kassel.de,
bernhard.grusie@gmx.de

Anne Keller[*]
University of Antwerp,
Dept. of Mathematics and Computer Science
Middelheimlaan 1
2020, Antwerpen, Belgium
anne.keller@ua.ac.be

ABSTRACT
With the ever increasing pervasiveness of software in every day's life, it is quite easy to explain children the importance of software development. Especially when using gadgets such as LEGO robots, one can fascinate young pupils. It is much harder though to find a fair link to the actual educational and research programs from a particular university without blowing the audience away with details of a particular Java framework. This paper illustrates how one can use Fujaba to involve children from 8 to 18 years old in realistic requirements elicitation workshops. The children implicitly get in touch with the object-oriented paradigm by playing in the real world the communication between objects in a robot's computer. Fujaba's visual object browser provides a convincing means to illustrate that the game adequately represents the robot's internals.

1. INTRODUCTION
Fujaba has been used for educational robotics programming since 2002 [2, 3]. The Fujaba NXT framework provides the technical infrastructure for interacting with the *NXT* version of the LEGO Mindstorms hardware [4]. So far, there has been more focus on the fine-tuning and debugging of this framework than on the elaborate use thereof. This paper describes the current state of the NXT framework and provides an overview of the quickly growing set of educational projects in which it has been used so far.

This work is based on a collaboration between the developers of the framework (from the University of Kassel) and external "users" thereof (from the University of Antwerp and the University of Eindhoven). From this collaboration, one can conclude that: (i) the framework supports smooth migrations to new implementation layers, and (ii) even when both teams use completely different robot hardware designs, one can share valuable software artifacts.

This paper also specifically points to the limitations of the NXT framework that was presented on the Fujaba Days in 2008 and indicates which problems were overcome and which tough issues remain unsolved so far. As a major novelty, this paper shows how to combine story driven modeling (SDM [1, 8]) with Statechart modeling for optimizing model readability. Obviously, such readability is an essential property of models that are used in an educational context. The remainder of this text is structured as follows: Section 2 briefly introduces the Fujaba NXT framework that is used throughout this paper. Section 3 presents a brief overview of the educational projects that use this framework. Section 4 describes which lessons have been learned from developing the NXT examples for these projects. Finally, Sections 5 and 6 close this paper by summarizing, concluding and pointing to future work. An extended version of this paper is publicly available as a technical report [6].

2. THE FUJABA NXT FRAMEWORK
This section briefly revisits those concepts from [4] that are essential for understanding the core of this paper.

The Fujaba NXT framework adds the ability to model complete applications for LEGO Mindstorms in Fujaba using model driven development with a graphical programming environment. It relies on the LeJOS [7] framework, a Java firmware and API for LEGO Mindstorms. The basic approach is to remotely control a NXT with a program running on a host PC. The main advantage of remote controlling the NXT is that it can be easily debugged with a Standard Java Debugger.

Using the eclipse Dynamic Object Browsing System (eDOBS), one can (i) inspect the java heap space and watch a visualization of objects, their attributation and links and (ii) interactively change attributes (or links) and invoke methods. This is very useful in the NXT context, as one can control the robot interactively by invoking motor methods manually within the context of the running program.

3. EDUCATIONAL PROJECTS
This section presents the educational projects in which the NXT library was applied. Each project was developed with a different target audience in mind. Therefore, for each project the strengths, weaknesses, opportunities and inherent limitations are evaluated.

[*]This work was partially funded by (i) the Interuniversity Attraction Poles Programme - Belgian State – Belgian Science Policy, project *MoVES*, and (II) the Research Foundation – Flanders (FWO) project G.0422.05. Additionally, the authors wish to thank Nick Baetens and Glenn van Loon for both their conceptual as well as practical work on the copy robot.

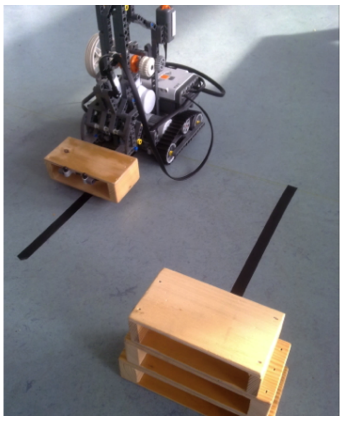


Figure 1: UniKassel Forklift robot in action

3.1 Towers of Hanoi: Fujaba Robotics Classic

The Towers of Hanoi example was first elaborated by Diethelm et al. [2, 3]. The 2002 solution relies on very limited LEGO hardware of the so-called *RCX* (Robotic Command Explorer) type. That RCX hardware has (among other problems) rather unreliable infrared PC communication. Since 2008, the NXT library relies on more modern hardware (e.g., bluetooth instead of infrared communication). In [4], the second author of this paper describes how this overcomes much of the problems from 2002. A remaining weak point of the 2008 solution is that it makes the robot drive blind just with the help of a single touch sensor. In the 2009 version, the robot uses straight black lines leading from the disc places to find each place. A single light sensor is used to detect that the robot crosses the black line while driving between the places. Figure 1 shows the robot with a disc picked up, following the black line on the ground.

Mind the so-called "continuous track" wheels, whose design is primarily known from caterpillar tanks. As discussed in [4], this wheel design provides a precise steering mechanism and the capability of turning on the spot, and is capable of carrying high loads. The hardware design has been improved further in 2009 [6].

3.2 Forklift in Factory

This example has been designed in the context of a promotion event at the University of Antwerp in 2008. During one week, the university welcomed secondary school classes in order to motivate them for higher education in exact sciences. The local Software Engineering groups working with Fujaba aimed to seize this opportunity for making students between the age of 14 and 18 excited about model-driven software development.

The goal of the role playing game is to derive behavioral models for an autonomous forklift robot. This robot needs to pick up all goods from a bill of materials and deliver them to an output line. More specifically, the robot needs to pick up four wheels, an engine, and a bodywork kit to enable the assembly of a car. In practice, students should mimic the different pieces of a robot (its navigator, its wheels, its sensors, ...) and walk through a classroom. Tables are arranged in rows that mimic the different shelves in the factory. At the end of each row, they can pick up an item. The robot is initialized next to the first row and should return there for item delivery.

3.3 A Copy Robot

The copy robot example was developed by students of the University of Antwerp in the course of a student project following the science week held at the University of Antwerp. The goal was to produce a educational demonstration introducing computer science principles to secondary school students in their last two years of school and consequently motivating them for computer science studies in general. The aim was to create a reusable teaching unit for occasions such as the prior science week. The result was a 1,5 hour interactive lecture that was held in a secondary school in Essen, Belgium. About 20 students attending the school's computer science class, between 16 and 17 years old, took part in the lecture.

The robot developed for this demonstration copies an image consisting of non-crossing, connected lines by scanning it with a light sensor and drawing it on a sheet of paper with a attached pen. The choice of a copy robot was motivated by the idea that copying and scanning are well known mechanisms that can be understood instantly. Additionally, the copy robot offers a physically compact setup (i.e., does not drive around) that is well suited for the intended interactive lecture setup (see Figure 6).

3.4 Wall•E Rescues Eve

Less than one year after the science week at the University of Antwerp, one of the instructors was asked to organize at Eindhoven University of Technology (TU/e) a science-related workshop for families with kids. It seemed promising to use this event as an opportunity to tackle some of the weaknesses of the Factory example described above. The so-called "open day" targeted kids between the age of 8 and 12 and a session should take about half an hour at most.

The following considerations have driven the design of a new Fujaba NXT example: (i) all existing examples were too complicated for the children younger than 12 years of age, (ii) the 2008 forklift design and its multi-step turning approach was a promising basis, (iii) several participants to the 2008 Factory Robot workshop had made enthusiastic references to the Wall•E movie.

Figure 7 shows an example execution of a Wall•E robot, controlled by a Fujaba NXT application. Note how the robot drives from the black line several times and backs up iteratively to find a state where his both light-sensors are on top of the black line again.

When arranging tables as a corridor in an S-shaped path, 12 year old kids need about 5 minutes to walk from start to end. In order to avoid cheating, one can blindfold particular role players.

3.5 Evaluation of the Examples

The main strengths of the Towers of Hanoi example (cfr., Section 3.1) are that (i) it has been implemented several times already (which facilitates a comparison between the different approaches), and (ii) it involves challenging tasks in the real world (picking up a block, delivering it at a variable height, moving from tower to tower, ...). An important weakness is that it requires an understanding of recursion. It turns out that very little children come to the intended solution spontaneously. Therefore, the Hanoi example will primarily be used as an internal testcase: from that perspective it presents the opportunity to use new sensor types (compass sensor for accurate 90-degree turns, ultrasonic, ...). Reusable functionality will obviously be moved to the appliation-independent library layer.

The forklift example (cfr., Section 3.2) is designed for students of 16 years or older. The example appeals especially to those that are concerned with industrially relevant education. The Copy Robot example (cfr., Section 3.3) has the advantage that the robot supports a detailed treatment of hardware related issues. A current limitation is that it is programmed in Java directly. This is due to the fact that it was programmed by students that were unfamliar with Fujaba. The fact that these students did not pick up the Fujaba modeling method autonomously did challenge us in improving the basic building blocks in the Fujaba NXT framework. Additionally, we are reminded of the importance of good documentation.

Since the Wall•E example (cfr., Section 3.4) has been developed most recently, we have been able to take into account the experience from performing sessions with the other examples [6]. The new example is designed to make kids from primary school enthusiastic about software engineering. About 80 children between 5 and 12 years of age have completed a survey that was designed to identify which parts of the educational session that children like most (and least) and which parts that children understand best (and worst). A key conclusion from this survey is that more time should (relatively) be spent with eDOBS: although quite a number children very much enjoy the role-playing game, a comparable amount of children wants to interact more with the actual robot. In fact, we have already experimented with some variants to the standard session setup. In one particular case, we noticed that a group of children already understood very much of the robot's internals after analyzing how the robot completed the scenario shown on Figure 7. Therefore, we skipped the role-playing and used eDOBS directly, with remarkable success.

4. LESSONS LEARNED

This section presents the main lessons that the authors have learned after the publication of [4]. We present new modeling guidelines and a list of known open issues.

4.1 Modeling Guidelines

The common approach to model a custom robot is to subclass one or more classes from the core API, like FNXT. This way, default behaviour can be easily overridden. Figure 2 shows an extract of the Fujaba NXT API: the FMotor and FTouchSensor class.

Some methods of these classes are synchronous whereas others are not. This is neccessary to model efficiently, but for inexpierenced developers, it is very difficult to distinguish between these. For example, all *waitFor...()*-methods

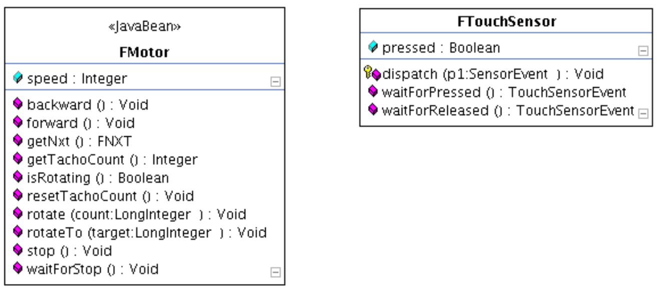

«JavaBean»
FMotor
speed : Integer
backward () : Void
forward () : Void
getNxt () : FNXT
getTachoCount () : Integer
isRotating () : Boolean
resetTachoCount () : Void
rotate (count:LongInteger) : Void
rotateTo (target:LongInteger) : Void
stop () : Void
waitForStop () : Void
FTouchSensor
pressed : Boolean
dispatch (p1:SensorEvent) : Void
waitForPressed () : TouchSensorEvent
waitForReleased () : TouchSensorEvent

Figure 2: Class diagram extract of the Fujaba NXT library layer.

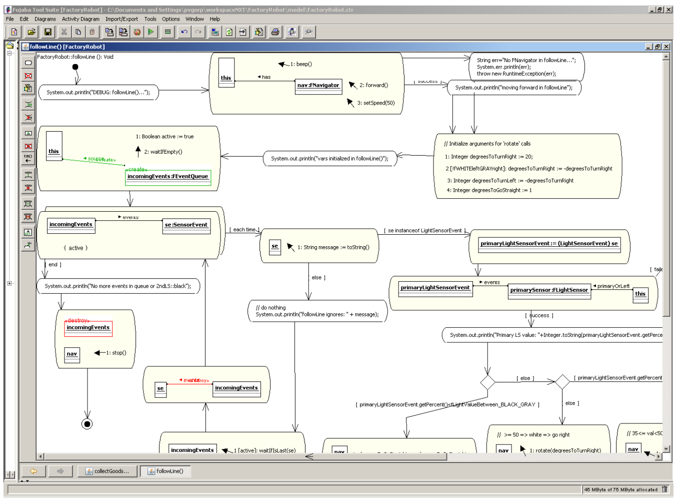


Figure 3: Example of SDM bad style.

are blocking, *forward()* and *backward()* not, whereas *rotate(long)* is synchronous as well: it returns when the rotate task is completed. A *waitFor...()* method implicitly encodes an event handler.

When using the *waitFor...()* methods frequently in story diagrams, one is scattering implicit event handlers. Figure 3 for example contains some, rather hard to find, *waitIfLast* and *waitIfEmpty* calls.

In general, Figure 3 illustrates the authors' negative experiences with the Story Driven Modeling of multi-threaded event handling. These experiences have eventually lead to the combined use of story diagrams and statecharts. Obviously, Figure 3 is not intended to be readable. In fact, it does not even fit on a mainstream notebook screen. One could work around this limitation by refactoring some fragments to separate methods. However, the core complexity, which relates to the explicit modeling of an event queue, cannot be removed using Story Driven Modeling constructs. Mind that this is not only an issue of forced over-specification (i.e, a lot of work) but more dramatically we have learned the hard way that this modeling style is very error-prone and hard to debug.

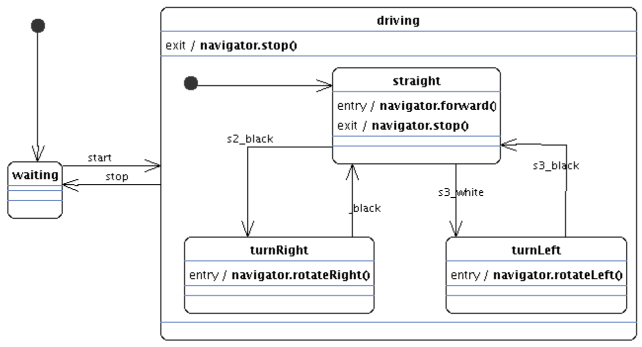


Figure 4: Line Follower algorithm as a statechart.

Statecharts seem to be a feasible solution to overcome the problems related to the Story Driven Modeling of event handling: each statechart runs in a separate thread implicitly while wait and notify methods are called when expected, behind the scenes. The framework ensures that all sensor events can be used to trigger state transitions. Currently, we use a very simple string encoding of the event triggers: <sensor-port> _ <state>, where sensor-port is s1, s2, s3 or s4 referring the hardware ports at the NXT brick, and state is *black*, *white*, *pressed* or *released*, depending on the connected sensor type.

Figure 4 shows the statechart of a common task for a driving robot: Follow a black line on the ground. The robot has two light sensors attached, which both should follow a wide black line. When one sensor goes off the line, it will read white, and an event trigger either *s2_white* or *s3_white* is fired. The statechart goes in one of the turn states and the robot will turn until both sensors read black again.

Recall from Section 3 that for sequential domain algorithms (e.g., let a robot collect goods from shelves) and basic navigation modeling (e.g., make a trike turn right), story diagrams do not expose the bad style that is discussed in the previous section. In fact, it does not make sense to replace the use of story diagrams by statecharts completely: in several cases, the pattern-based specification approach and the use of icons appears a perfect fit for all children between 8 and 18 years of age. Therefore, a multi-formalism modeling approach seems most appropriate.

Also mind that we decided only show to the children those diagrams that illustrated good modeling style (see [6] for examples). In the authors' opinion, it is not yet relevant for the children to learn from advanced pitfalls in the context of sessions that are primarily intended to stimulate motivation and a basic understanding.

4.2 Open Issues

Work on the Fujaba NXT framework and the construction of the examples described in this paper has so far been quite challenging. Although this should no longer be visible in a finished example, it does have an impact on the number of examples that has been completed successfully. The authors are convinced that a core issue has been tackled with the new statechart approach. However, the reader should be aware that other challenging issues are still open. Therefore, this section provides an overview of such issues and some known workarounds. Although these issues have not been solved yet, new developers of Fujaba NXT examples can save time by taking into account the known dangers and pitfalls, as experienced by the authors.

- when debugging Fujaba NXT applications (or embedded software in general), one can often not easily make erroneous behavior available for replay by other developers. This is due to the large amount of electrical components that can have bugs or thoughput limitations too (bluetooth dongles, electric cables, ...). So far, the first and second author managed to collaborate remotely by means of video sharing. In the case that the other developer could not reproduce the undesirable behavior, the first developer understood he had to consider deploying on another machine before spending more time on software debugging. In future work, best practices for using the simulation adapters will be investigated to make the debugging cycle more reliable and less time consuming.

- debugging multi-threaded software is a challenging task in general. The following characteristics make debugging the multi-threaded Fujaba NXT applications particularly hard to debug: when suspending a thread running on the host computer, the related threads on the LEGO components may continue to execute. To tackle this, special debugging features should be added to the Fujaba NXT framework (e.g., stopping all motors when suspending the VM on a breakpoint). This can be tightly integrated with Design Level Debugging [5]. Also mind that Design Level Debugging facilities would be very valuable too for the Statechart models.

- The underlying LeJOS framework still exposes bugs (obviously where they are least expected) and has a counter-intuitive design. For example, the bluetooth startup code is in the static initializer of the Motor class.

5. SUMMARY AND CONCLUSIONS

This paper illustrates how the Fujaba NXT framework is applied with reasonable success in several educational projects. An intended outcome of these projects is that the children acknowledge that role playing can provide a useful basis for programming. The underlying hypothesis is that this will break perceptions (if any) of software development as a pure asocial activity and we hope this attracts additional bright people to the software industry that would otherwise have choosen another path.

The framework consists of (i) Fujaba models that provide some basic data structures and building blocks of primitive sensor and motor functionality, and (ii) a set of hand-coded wrapper classes that integrate third party (LeJOS) binaries. The object-oriented nature of the Fujaba models enables one to easily apply inheritance and delegation techniques to deal with variability in robot hardware designs (e.g., continuous track designs versus the trike design). The underlying LeJOS framework does not have this characteristic.

The Fujaba NXT framework turns out to be useful for role-playing sessions (at least) with kids between 8 and 18 years old. From 2002 onwards, behavioral modeling was done with story diagrams that generalized the behavior of story boards, that in turn represented snapshots from a role playing game [2]. The authors of this paper still acknowledge that the unique strengths of Story Driven Modeling remain (i) its semi-structured method for role-playing, (ii) the simple mental mapping for role players, since several domain alogithms involve rewriting steps, (iii) its syntax specializability by means of a simple icon mechanism, and (iv) its tool support for runtime visualization (eDOBS).

In 2008, the inherent limitations of this approach became better understood and since 2009 one can bypass these limitations by expressing event-based behavior using Statecharts.

6. FUTURE WORK

Section 4.2 already lists the technical issues that remain to be solved. This section focuses on future work on the conceptual level.

From the Copy Robot example, we learn the following: instructors cannot expect that bachelor or master students without Fujaba training will use story diagrams (or statecharts) spontaneously for the development of new Fujaba NXT projects. Such students do rely on a story boarding process but resort to pseudocode for generalizing method behavior. The authors consider it a challenge for the Fujaba community to provide more, and more elaborately documented, examples of story diagrams (and statecharts). The authors themselves will continue their collaboration to contribute such examples to the community. On the sort term, the Fujaba NXT library will be extended with more ready-to-reuse basic functionality.

Additionally, some university student projects related to Fujaba NXT are being supervised and even more will be supervized on the longer term. These projects should result in (i) stable hardware/robot/sensor setups (with building instructions) and (ii) multiple modeling solutions for one given problem and one or more robot designs.

7. REFERENCES

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

[2] I. Diethelm, L. Geiger, A. Zündorf. UML im Unterricht: Systematische objektorientierte Problemlösung mit Hilfe von Szenarien am Beispiel der Türme von Hanoi. *Erster Workshop der GI-Fachgruppe Didaktik der Informatik, Bommerholz, Germany*, Oct. 2002.

[3] I. Diethelm, L. Geiger, A. Zündorf. Fujaba goes Mindstorms. *Objektorientierte Modellierung zum Anfassen; in Informatik und Schule (INFOS) 2003, München, Germany*, Sept. 2003.

[4] R. Jubeh. Simple robotics with Fujaba. In *Fujaba Days*. Technische Universität Dresden, Sept. 2008.

[5] Leif Geiger. Design Level Debugging mit Fujaba. In *Informatiktage*, Bad Schussenried, Germany, 2002. der Gesellschaft für Informatik.

[6] P. Van Gorp, R. Jubeh, B. Grusie, and A. Keller. Fujaba hits the Wall(-e) – Beta working paper 294, Eindhoven University of Technology. `http://beta.ieis.tue.nl/node/1487`, Nov. 2009.

[7] LeJOS, Java for Lego Mindstorms. http://lejos.sourceforge.net/, 2009.

[8] A. Zündorf. Story driven modeling: a practical guide to model driven software development. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 714–715. ACM, 2005.

Figure 5: Pictures from the Science Week event in Antwerp, 2008.

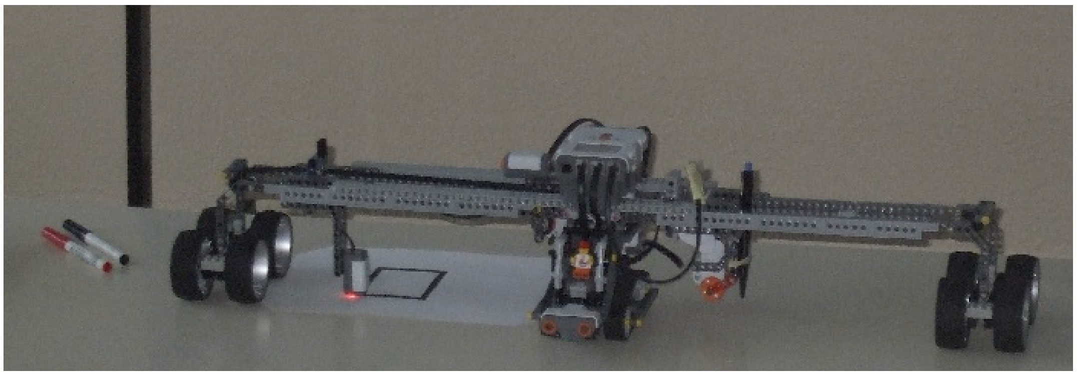


Figure 6: Copy Robot in action.

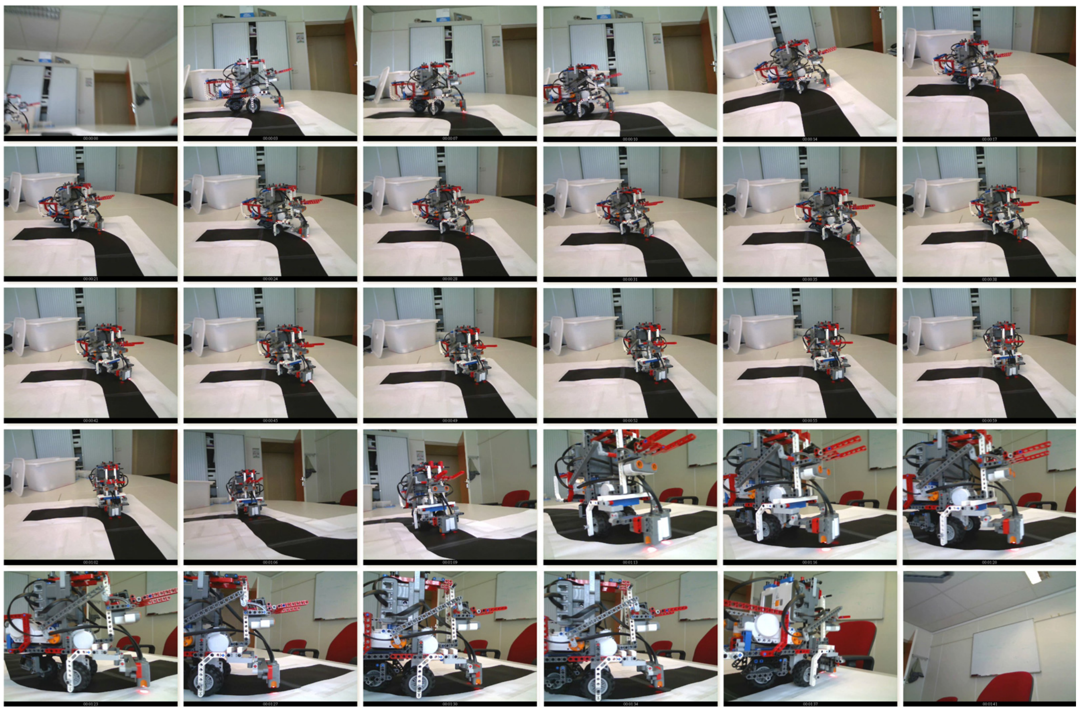


Figure 7: Example run of the Wall•E Rescues EVE scenario.

Synthesis of Component Behavior [*]

Tobias Eckardt, Stefan Henkler
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany

[tobie|shenkler]@uni-paderborn.de

ABSTRACT

Component-based architectures are widely used in the domain of embedded real-time systems. For managing complexity and improving quality of component-based architectures, separation of concerns is one of the most important principles. For one component, separation of concerns is realized by defining the overall component functionality by separated protocol behaviors. One of the main challenges of applying separation of concerns is the later (application specific) composition of the separated, maybe interdependent concerns. This is also one of the main disadvantages of current component-based approaches as at least the separated specification and automatic integration of interdependent concerns is not supported. Moreover, the overwhelming complexity of embedded systems, which is especially present if these systems are also of distributed real-time character, requires to also consider safety requirements for the composition of the separated concerns. We present an approach which addresses these problems by a well-defined automatic composition of protocol behaviors with respect to interdependent concerns specified as composition rules. The composition is performed by taking a proper refinement relation into account so that analysis results of the separated concerns are preserved which is essential for safety critical systems.

1. INTRODUCTION

Component-based architectures are widely used in the domain of embedded real-time systems. The main benefits of using components are their support for information hiding and reuse. The interface of a component is well defined by structural elements and a collaboration of protocols (cf. [BSW00]). The overall component behavior is defined by the (parallelly executed) protocol behaviors. Dependencies between components are reduced to the knowledge of interfaces or ports. Thus, a component can be exchanged if the specified port remains fulfilled. The port and interface definitions of architectural components therefore facilitate the construction of complex functionality by the composition of existing components.

For managing complexity and improving quality of component-based architectures, separation of concerns [Dij76] is one of the most important principles. It enables primary software engineering goals like adaptability, maintainability, extendability and reusability. Accordingly, advanced applications of separation of concerns have gained popularity like aspect-oriented programming (AOP) [KLM97], for example.

One of the main challenges of applying separation of concerns is the later (application specific) composition of separated, maybe interdependent concerns [TOHS99]. In general, we can distinguish between structural, data, and behavioral composition. In the area of structural composition, approaches exist for example, that consider the software architecture as well as architectural patterns [BMR$^+$96, GP95]. For data composition approaches like [Gru93] support the generation of suitable translators. In [Mil89, GV06] approaches for the behavioral composition are presented. The overwhelming complexity of embedded real-time systems, however, requires to also consider safety requirements for the composition which is not included in these approaches. On the other hand, component-based approaches for embedded real-time systems [Sys07, UML08, GS03, JS05, MRBC03, Sel96] suffer the support for interdependent concerns for the well-defined composition.

In this paper, we present an approach which addresses these problems by a well-defined automatic composition of protocol behaviors with respect to interdependent concerns specified as composition rules. The composition is performed by taking a proper refinement relation into account so that analysis results of the separated concerns are preserved which is essential for safety critical systems. For this, we extend our modeling approach MECHATRONIC UML which addresses the development of complex embedded real-time systems. MECHATRONIC UML supports the specification and compositional verification of real-time coordination by (1) applying component-based development and pattern-based specification [GTB$^+$03] and (2) the integrated description and modular verification of discrete behavior and continuous control of components [GBSO04]. In this paper we give a brief overview of the approach and the integration in the Fujaba Real-Time Tool Suite[1]. For more details, we refer to [Eck09].

In the following section, we present a case study which is used to exemplify our approach. A sketch of our approach is presented in Section 3. In Section 4, we present the concept of composition rules. The synthesis algorithm is described in Section 5. Related work is discussed in Section 6. We conclude with a summary and future work in Section 7.

2. CASE STUDY

The research initiative *Neue Bahntechnik Paderborn (NBP – New Rail Technology Paderborn)*[2] is developing a rail-based transportation system at the University of Paderborn since 1997. This system

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

[1]http://wwwcs.uni-paderborn.de/cs/fujaba/projects/realtime/index.html
[2]http://www.railcab.de

requires most modern techniques of mechatronics – mechanical engineering, electrical engineering and software engineering. The basis of the system are driverless vehicles, called *RailCabs*, which travel on demand without a fixed schedule and are able to carry either passengers or goods. The system and especially the techniques for developing mechatronic systems are studied with intense effort by the *Collaborative Research Centre 614 – "Self-optimizing Concepts and Structures in Mechanical Engineering" (CRC 614)*.

Compared to conventional railway systems, RailCabs have two main advantages: (1) They can travel in *automatic convoy operation* mode and (2) they have a built-in *active track guidance* system. Traveling in *automatic convoy operation* mode, RailCabs are able to build convoys dynamically, without physical contact between each other. This increases traffic flow and minimizes energy consumption. With the *active track guidance* system, each RailCab is able to steer its wheels in order to adapt to the actual track trace and condition. This increases driving comfort, decreases wear out and makes passive switches possible.

For the active track guidance system, the RailCab needs track information in advance, in order to be able to steer its wheels correctly. To receive this track information, the RailCab can either communicate with an external entity or measure the track itself using its internal track sensors. In this example, the external entity is represented by so-called *base stations*. When receiving the track information in advance via a base station, the RailCab can achieve the best comfort but also has to establish and uphold a connection to a base station which might not always be possible. Using its internal track sensors instead, the RailCab is independent, but can only adapt to those track characteristics it is able to process in the short period of time between receiving the signal input of the sensor and the wheel actually being on that piece of track. A simple example of four RailCabs RC1 to RC4 and four base stations BS1 to BS4 is shown in Figure 1 where each RailCab is connected to the nearest base station.

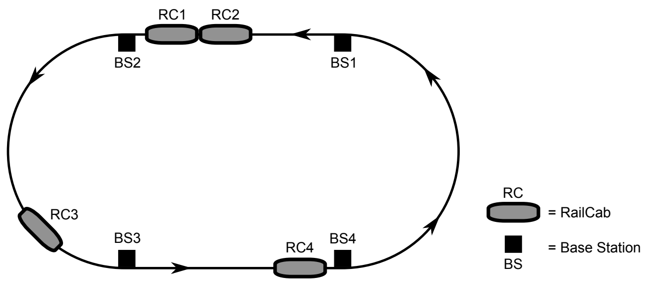


Figure 1: RailCabs and Base Stations on a Track

For traveling with automatic convoy operation, a RailCab needs information about other RailCabs driving nearby, which constitute potential convoy partners. In this example, this information is obtained from the base station the RailCab is connected to. Once a convoy partner is found, the (in driving direction) rear RailCab contacts the front RailCab and proposes to initiate a convoy. The front RailCab is able to accept or reject the convoy, depending on its schedule or other factors. If the front RailCab accepts the convoy, it has to do so within a certain time interval and communicate this to the rear RailCab. This can then again start the convoy operation by driving into the slipstream of the front RailCab and by reconfiguring its speed control unit from a velocity based mode to a distance based mode. This reconfiguration is necessary because driving at the exact same velocity is very difficult and only slight differences in the velocity of two RailCabs driving in a convoy may cause collisions within seconds. Therefore, the rear RailCab has to measure the distance to the front RailCab and adjust its velocity according to that value. As a collision can lead to high financial damage or even life threatening situations, the convoy operation contains safety critical aspects with hard real-time requirements. In Figure 1 RailCabs RC1 and RC2 operate in convoy, where RC1 is the front and RC2 the rear RailCab.

3. APPROACH

In MECHATRONIC UML separation of concerns is realized by applying *component based development* and in accordance with that by rigorously separating inter-component from intra-component behavior. Following this concept, the system is decomposed into participating components and *real-time coordination patterns* [GTB+03], which define how components interact with each other.

To exemplify this using the case study, we specify two components BaseStation and RailCab (Figure 2) and two coordination patterns Registration and Convoy, which define the before described communication behavior between RailCabs and base stations.

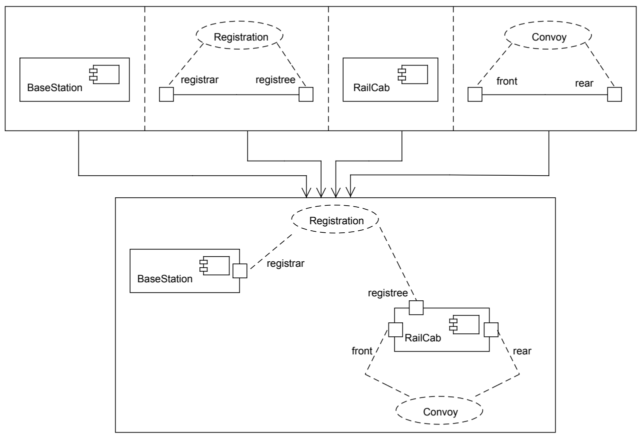


Figure 2: Combining Separate Specifications in MECHATRONIC UML

In real-time coordination patterns, *roles* are used to abstract from the actual components participating in one coordination pattern. This way, it is possible to specify and verify coordination patterns independently from other coordination patterns and component definitions and therefore to reduce complexity. In Figure 2 the participating roles of the Registration pattern are registrar and registree; the roles of the Convoy pattern are front and rear. Each role behavior is specified by one protocol statechart. The statecharts of the rear role and the registree role are depicted in the screenshots in Figure 3 and 4 which also show the realtime statechart editor integrated in the Eclipse version of the Fujaba Real-Time Tool Suite. The statecharts for the front role and the registrar role only form corresponding counterparts and are therefore not depicted.

To obtain an overall system specification later in the development process, the separated components and coordination patterns have to be combined again (Figure 2). The problem which inherently arises at this point is that separate parts of the system were specified as independent from each other when they are in fact not. This

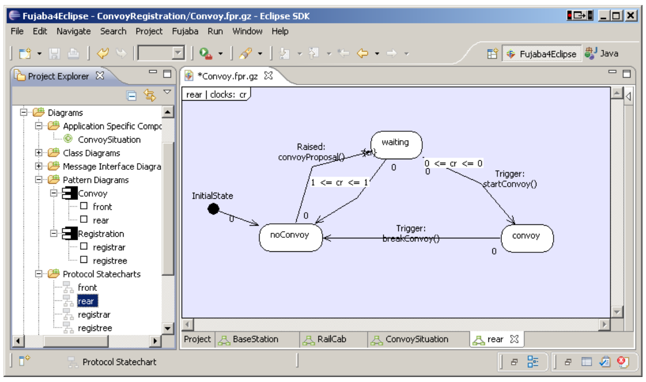


Figure 3: Rear Role Protocol Statechart

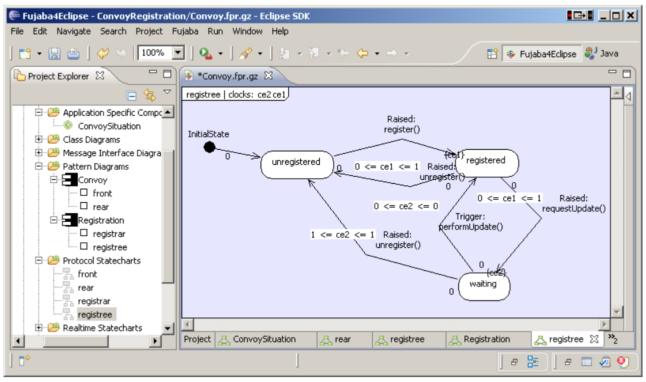


Figure 4: Registree Role Protocol Statechart

means that during the process of combining the separate parts of the system, additional dependencies between the particular specifications have to be integrated. At the same time, the externally visible behavior of the particular behavioral specifications may not be changed in order to preserve verification results [GTB$^+$03].

In the overall system view of the RailCab example (Figure 2), the RailCab component takes part in both, the Registration and the Convoy pattern. While those patterns have been specified independently from each other, a system requirement states that in convoy operation mode, each participating RailCab has to be registered to a base station. Accordingly, a dependency between both patterns exists, when applied by the RailCab component. As a result, the behavior of the registree role and the behavior of the rear role have to be refined and synchronized with each other when applied by the RailCab component in order to fulfill the system requirements. Still, it has to be regarded that the externally visible behavior of the RailCab component does not change. If this process of refinement and synchronization is performed manually, it is a time consuming and error-prone process. Consequently, this implies the necessity for automation in order to guarantee the required quality of the developed systems.

In the proposed approach, we separate the specification of dependencies and the specification of the pattern role behaviors in order to perform an automatic synthesis for the overall component behavior. Once the synthesis is performed, it is further checked if the synthesized component behavior refines each of the particular pattern role behaviors properly.

The approach requires (1) the definition of a suitable refinement relation for (real) dense time systems, (2) the employment of a suitable and efficient abstraction of the timed behavioral models which is needed to perform the refinement check and (3) the integration of reconfiguration behavior. The result is a fully automatic synthesis algorithm where dependencies between separate behavioral specifications are specified explicitly by so-called *composition rules* (cf. [TOHS99]). Accordingly, the input for the algorithm are composition rules and separate behavioral specifications (Figure 5) in the form of MECHATRONIC UML's realtime statecharts. If the synthesis is possible without violating the externally visible behavior of any of the input specifications, the output is one parallelly composed component behavior which combines all of the input specifications as well as the composition rules. If the synthesis is not possible, the algorithm returns a conflict description indicating the reason for the impossibility.

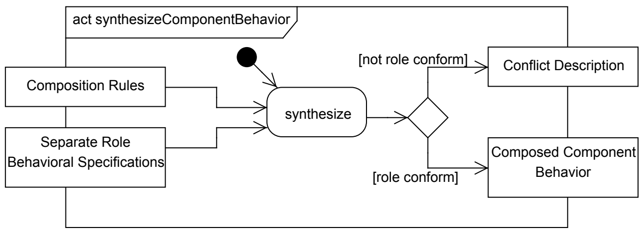

act synthesizeComponentBehavior
Composition Rules
Separate Role Behavioral Specifications
synthesize
[not role conform]
Conflict Description
[role conform]
Composed Component Behavior

Figure 5: Activity Diagram Illustrating the Basic Synthesis Approach

With this algorithm system developers are able to attach composition rules to components of their views to specify dependencies to other views. These composition rules are later factored in automatically during the composition of the full system specification by the synthesis algorithm if none of the behavioral specifications is violated. If a behavioral specification is violated, the developer can use the conflict description to find the part of the specification which causes the violation in order to resolve the conflict. By giving the dependencies explicitly by means of composition rules, a detailed conflict description can be obtained referring to the conflicting composition rules. This way, the manual interaction during the composition of the views is reduced to the essential parts: (1) specifying dependencies as composition rules, (2) analyzing conflict descriptions and (3) resolving these conflicts. The second and the third action however, are only necessary if a conflict actually exists between the input specifications.

4. COMPOSITION RULES

With composition rules, interdependent concerns for the separate role behaviors can be specified as system properties which synchronize parts of the separated role behavioral models. Generally speaking, system properties can be specified in terms of safety and liveness properties for a given behavioral specification [Lam77, Hen92]. *Safety properties* state that something bad will never happen during the execution of a program. *Liveness properties* state that something good will happen eventually. Transferring this to the context of automata synchronizations, these properties always concern two or more automata. Consequently, a *safety property for synchronization* states that something bad will never happen, when executing the corresponding automata in parallel, while a *liveness property for synchronization* expresses that something good will eventually happen during this parallel execution.

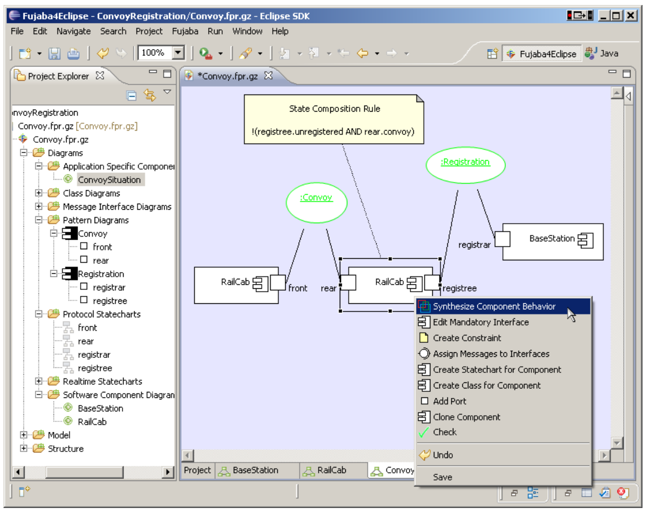
Transferring these properties to composition rules, we are able to specify both safety and liveness properties. Safety properties can be specified (1) by means of *state composition rules* in terms of forbidden state combinations of the parallel execution and (2) by means of *event composition automata* by adding further time constraints to time guards of selected transitions. Liveness properties in turn can be specified through state composition rules and event composition automata by adding further time constraints to location invariants of location combinations of the parallel execution. As we integrate controller configurations in states, we can describe dependencies between controllers or controller configurations by the aforementioned composition rules, too.

In the screenshot of the component diagram editor (Figure 6) the annotated state composition rule is given with r_1, where

$$r_1 \quad = \quad \neg((unregistered, true) \wedge (convoy, true)).$$

r_1 formalizes the pattern overlapping system requirement explained in Section 3. Correspondingly, it defines that a RailCab is not allowed to rest in states $(unregistered, true)$ and $(convoy, true)$ at the same time, where the clock constraint $true$ denotes that all clock values of the corresponding automata are concerned.

Due to the lack of space, we omit the description and illustration of event composition automata and instead refer to [Eck09].


Fujaba4Eclipse - ConvoyRegistration/Convoy.fpr.gz - Eclipse SDK

State Composition Rule

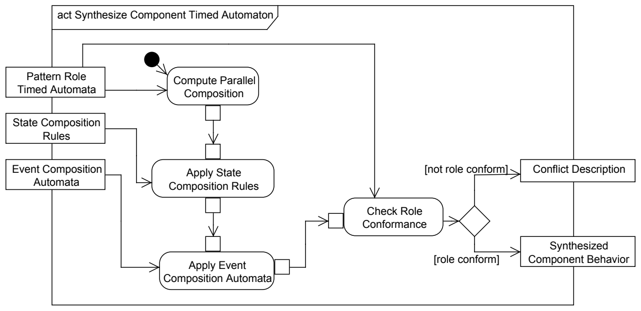
!(registree.unregistered AND rear.convoy)

Figure 6: Specified State Composition Rule and the Menu Entry to Synthesize the Component Behavior

5. SYNTHESIS ALGORITHM

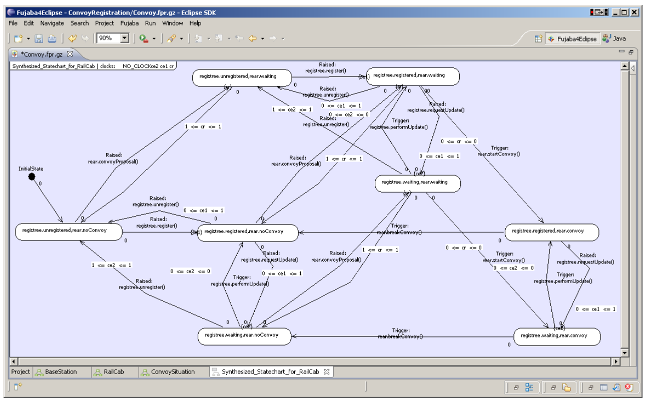
The input for the synthesis algorithm are both the composition rules and the role behaviors as MECHATRONIC UML's real-time statecharts [BGHS04] or reconfiguration charts [GBSO04], respectively. As the behavioral models for the pattern roles are based on the semantics of timed automata [AD90, HNSY92], they can also be transformed into semantically equivalent timed automata. Consequently, the complete synthesis procedure is performed on the basis of timed automata and we refer to the transformed role statecharts as pattern role automata.

The synthesis algorithm is divided into four distinct steps (see Figure 7), which are described in detail in [Eck09]. First, the parallel composition of the role automata is computed, which forms an explicit model for the parallel execution of the pattern role automata. On this parallelly composed timed automaton the composition rules are applied, by removing the forbidden system states specified by the state composition rules and by including the specified event composition automata in the parallelly composed automaton. In the last step, it is verified that the externally visible behavior of the particular role automata is preserved, as the changes made on the parallelly composed automaton by means of the appliance of composition rules might lead to violations of properties of the original role behaviors. In Figure 8 the synthesized behavior of the shuttle component with the composition rule as shown in the last section is depicted.



act Synthesize Component Timed Automaton
Pattern Role Timed Automata
State Composition Rules
Event Composition Automata
Compute Parallel Composition
Apply State Composition Rules
Apply Event Composition Automata
Check Role Conformance
[not role conform]
[role conform]
Conflict Description
Synthesized Component Behavior

Figure 7: Synthesis Algorithm for Timed Automata



Fujaba4Eclipse - ConvoyRegistration/Convoy.fpr.gz - Eclipse SDK
Synthesized_Statechart_for_RailCab
InitialState

Figure 8: Synthesized Component Behavior of the RailCab Component

6. RELATED WORK

The field of controller synthesis [AMP95, AT02, GGR08] deals with the problem of synthesizing a behavioral model for a *controller* which interacts with some *environment*. In a controller, interaction is specified through alternating actions between the controller and the environment. Consequently, for the behavioral model a special type of timed automaton, a *timed game automaton* [AMP95], is applied. In a timed game automaton, transitions are partitioned into those controllable by the controller and those controllable by the environment. The main difference to our synthesis approach is that the given behavioral model of controller synthesis does not take a compositional character of this model into account as this is not necessarily given in the underlying controller behavior. In our approach this is given by the independent pattern role automata. Consequently, the compositionality can also not be considered for the specification of the properties which have to be synthesized. Altogether, this results in a different equivalence relation between the original and the synthesized model which in turn results in different synthesis algorithms.

In [GV06] a synthesis procedure for the behavior of interacting components is presented. [Sei07] extends this approach by the notion of time concerning the behavioral models and the state restrictions. The main difference between our approach and [Sei07] approach is that in [Sei07] a discrete abstraction of time is applied. This is not applicable for MECHATRONIC UML. Indeed, discrete time approaches are not well suited and in general not applicable for embedded systems [CGP00]. Furthermore, [Sei07] applies both the parallel composition and the state restrictions to the internal discrete-time automaton model, while we use the abstraction only to verify role conformance and apply the composition rules directly to the timed automaton model instead. At last, [Sei07] also allows for representational non-determinism by the explicit use of τ-transitions representing internal component behavior, instead of treating the behavior of other ports as internal component behavior. Consequently, the applied refinement relation differs in this point. Nevertheless, [Sei07] proposes to allow an arbitrary number of delay transitions and internal component behavior transitions between the relevant actions of the port automata, which is equally applied in our refinement relation.

7. CONCLUSION AND FUTURE WORK

In this paper we proposed an approach to automatically synthesize the behavior for a MECHATRONIC UML component which takes part in several real-time coordination patterns. The current approach of MECHATRONIC UML suffers from the manual refinement and synchronization of the coordination role behaviors which has to be accomplished to construct the component behavior. Therefore, we propose to specify dependencies between several role behaviors separately by means of composition rules. Additionally, we defined a procedure to automatically integrate the composition rules for a given set of role behaviors. Afterwards it is checked that the resulting component behavior refines each of the role behaviors properly. For future work we plan to perform a complete evaluation of the approach regarding a realistic set of case studies. This way, it could also be evaluated if the proposed composition rule formalism is sufficient to specify existing dependencies between several coordination roles.

8. REFERENCES

[AD90] Rajeev Alur and David L. Dill. Automata for Modeling Real-time Systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science (LNCS)*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[AMP95] Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, pages 1–20, London, UK, 1995. Springer-Verlag.

[AT02] Karine Altisen and Stavros Tripakis. Tools for controller synthesis of timed systems. In Paul Pettersson and Wang Yi, editors, *Proceedings of the 2nd Workshop on Real-Time Tools (RT-TOOLS'02)*, August 2002.

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental design and formal verification with UML/RT in the FUJABA real-time tool suite. In *Proc. of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, pages 1–20, October 2004.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture* , volume 1. John Wiley & Sons, 1996.

[BSW00] Jan Bosch, Clemens A. Szyperski, and Wolfgang Weck. Component-oriented programming. In Jacques Malenfant, Sabine Moisan, and Ana M. D. Moreira, editors, *ECOOP Workshops*, volume 1964 of *Lecture Notes in Computer Science*, pages 55–64. Springer, 2000.

[CGP00] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, January 2000.

[Dij76] E.W. Dijkstra. *A discipline of programming*. Prentice-Hall Series in Automatic Computation, 1976.

[Eck09] Tobias Eckardt. Synthesis of reconfiguration charts. Diploma Thesis, Software Engineering Group, University of Paderborn, Oct 2009. to appear.

[GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver 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.

[GGR08] Stephanie Geist, Dmitry Gromov, and Jörg Raisch. Timed discrete event control of parallel production lines with continuous outputs. *Discrete Event Dynamic Systems*, 18(2):241–262, 2008.

[GP95] David Garlan and Dewayne Perry. (introduction to the) special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, 1993.

[GS03] Gregor Gössler and Joseph Sifakis. Component-based construction of deadlock-free systems. In *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science*, volume 2914/2003 of *Lecture Notes in Computer Science*, pages 420–433. Springer Berlin / Heidelberg, 2003.

[GTB+03] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan 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.

[GV06] Holger Giese and Alexander Vilbig. Separation of non-orthogonal concerns in software architecture and design. *Software and System Modeling (SoSyM)*, 5(2):136 – 169, 6 2006.

[Hen92] Thomas A. Henzinger. Sooner is Safer than Later. *Information Processing Letters*, 43(3):135–141, 1992.

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS)*, pages 394–406. IEEE Computer Society Press, 1992.

[JS05] Ethan K. Jackson and Janos Sztipanovits. Using separation of concerns for embedded systems design. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 25–34, New York, NY, USA, 2005. ACM.

[KLM97] *Aspect-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1997.

[Lam77] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MRBC03] Mohammad Reza Mousavi, Michel Reniers, Twan Basten, and Michel Chaudron. Separation of concerns in the formal design of real-time shared data-space systems. In *ACSD '03: Proceedings of the Third International Conference on Application of Concurrency to System Design*, page 71, Washington, DC, USA, 2003. IEEE Computer Society.

[Sei07] Andreas Seibel. Behavioral Synthesis of Potential Component Real-Time Behavior. Diploma Thesis, Software Engineering Group, University of Paderborn, June 2007.

[Sel96] Bran Selic. Real-time object-oriented modeling (room). In *2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96), June 10-12, 1996, Boston, MA, USA*, pages 214–. IEEE Computer Society, 1996.

[Sys07] *OMG Systems Modeling Language (SysML) Specification, Version 1.0*, 2007. SysML07.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, New York, NY, USA, 1999. ACM.

[UML08] *UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE), Version Beta 1.0*, 2008. MARTE08.

Modeling Safe Reconfiguration with the FUJABA Real-Time Tool Suite [*]

Claudia Priesterjahn, Matthias Tichy
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[cpr|mtt]@uni-paderborn.de

ABSTRACT

Software systems are increasingly built to exhibit self-* properties (e.g. self healing or self optimization) which require reconfiguration and change at runtime. This is even true for embedded or mechatronic systems which are often used in safety critical environments. In those cases, the effects of the reconfiguration on the safety of the system must be carefully analyzed. We present an approach to ensure the safety of self-* systems during runtime by checking whether a reconfiguration is allowed w.r.t. the hazard probability and the associated damage after the reconfiguration. The approach has been implemented as plugins for the Fujaba Real-Time Tool Suite.

1. INTRODUCTION

Embedded systems are often used in a safety-critical context. Consequently, hazards and risks have to be considered during system development. Standard development approaches (cf. [Lev95]) for safety-critical computer systems require hazards to be identified and the associated damage to be defined. The damage is the result when a hazard leads to an accident. We define risk as the product of the hazard probability and the damage of the accident linked to the hazard. We refer to [Lev95, Sto96] for more detailed definitions.

Self-* systems pose a challenge to these activities as they change their behavior and structure during runtime. This leads to changes of the hazard probabilities and damage values, which in turn affect the associated risks. However, this also opens up possibilities for risk management as it enables to adapt the behavior, and thus the hazard probability, by reacting to changes in the damage during runtime. Consequently instead of checking whether the *worst case damage* results in a safe risk for all configurations before runtime, we check during runtime whether the target configuration of a reconfiguration activity is safe w.r.t. the current damage.

A recent approach on risk analysis computes risk component wise and combines the results for a static component structure [YA02]. In contrast to our work, Yacoub and Ammar do not address self-* systems with different configurations during runtime.

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

In previous works [GTS04, GT06, THMvD08], we have presented a component-based hazard analysis approach which also considers the associated risk. This approach is specifically tailored to self-* systems which change their behavior by structural adaptation.

The approach extends all components by an abstract failure propagation which relates random errors in the components to failures at the components' ports. We follow the terminology of Laprie [Lap92] by associating *failures* – the external visible deviation from the correct behavior – to the ports where the components interact with their environment. *Errors* – the manifestation of a *fault* in the state of a component – are restricted to the internal of the component. We use Boolean logic with quantifiers to formally encode the failure propagation of the system and the occurrence of hazards.

The failure propagation models of all components in the system structure are then combined to form the system failure propagation. This system failure propagation is analyzed w.r.t. the errors which have to manifest so that a given hazard occurs. The hazard probability is computed based on the individual probabilities of the errors. Finally, the risk is the product of hazard probability and damage.

For the special case of self-* systems which change their structure during runtime, the approach supports to compute all structural configurations. These configurations are also considered in the aforementioned analysis. We determine the configurations in which a hazard can occur and, quantitatively, the configurations with the highest and worst hazard probability and risk. We currently pessimistically abstract from different damage values during runtime, which would result in different risk values, by employing the worst case damage.

In this paper, we present how we deal with damage values which change during runtime. The basic idea is that we refine the structural adaption behavior of the system components by additional checks whether the reconfiguration to another structure is allowed with respect to the current damage value and the defined maximum risk.

In our approach, the reconfiguration behavior is distributed over the system components but the hazard probability depends on the system structure which is typically not known by the individual components. Therefore, we opted to add a central component called the Risk Coordinator. This component is asked by a component whether a reconfiguration is safe with respect to the risk and allows or disallows this reconfiguration.

We exemplify our approach based on a scenario from our real-life RailCab project. The RailCab project[1] was founded at the University of Paderborn in 1998 in order to develop a new railway system which features the advantages of both public and individual transport in terms of cost and fuel efficiency as well as flexibility and comfort. The novel system is characterized by autonomous vehicles operating on demand instead of trains being determined to a fixed schedule. One particular goal of the RailCab project is to reduce the energy consumption due to air resistance by coordinating the autonomously operating RailCabs in such a way that they build convoys whenever possible. The RailCabs use different configurations w.r.t. whether they are driving alone or in a convoy. In the first case, they use a speed controller whereas they use a sophisticated distance controller in the second case. The damage associated to a hazard can change during runtime, e.g. in response to loading and unloading of goods like hazardous materials.

We present our approach in the following section. This presentation focuses on the behavioral extension of the reconfiguration behavior taking the real-time characteristics of embedded systems into account. In Section 3 we sketch how the presented approach has been implemented in the Fujaba4Eclipse Real-Time Tool Suite. Finally, we give a conclusion and an outlook on future work.

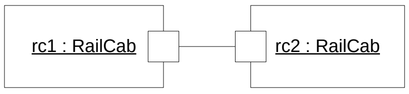
2. SATISFYING RISK CONSTRAINTS DURING RECONFIGURATION

In order to avoid configurations that exceed the maximum risk we enhance our system by a central component that performs risk analysis for the requested configuration to allow or prohibit reconfiguration. Therefore, we first introduce the extension of the architecture before we present the extension of the behavior models that is necessary to enable the blocking of reconfiguration transitions.

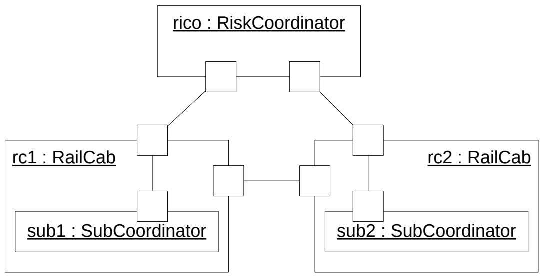
2.1 Architecture Extension

The system architecture is extended by an additional component which supervises the reconfigurations of the system, namely the *Risk Coordinator*. Before executing a reconfiguration each components sends a request to the Risk Coordinator. The Risk Coordinator computes the risk of the total system after the requested reconfiguration and checks if the system would still satisfy the maximum risk level after this reconfiguration. Depending on this verdict the Risk Coordinator allows or prohibits the reconfiguration. Further, each system component contains a sub component *Sub Coordinator* that encapsules the communication between this component and the Risk Coordinator. Figure 1(a) shows an examplary system consisting of two RailCabs. Figure 1(b) shows the same architecture extended by a Risk Coordinator instance and a Sub Coordinator instance for each RailCab Instance.

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



(a) Architecture of Two RailCabs



(b) Architecture of Two RailCabs with a Coordinator Component

Figure 1: Architecture Extension

2.2 Behavior Extension

We define the component's behavior by hybrid Reconfiguration Charts - UML state machines extended by time, continuous behavior and reconfiguration. Reconfigurations are specified by embedding configurations into the states. A reconfigation can result in a different hazard probability for the system or a different damage value.

We introduce *Safety Transitions* as an extension of hybrid Reconfiguration Charts. Safety Transtions can be blocked in order to prevent unsafe configurations. Figure 2 shows an example of a Safety Transition between the states *noConvoy* and *convoy* representing the reconfiguration taken in order to join or build a convoy of RailCabs. Safety Transitions are drawn as a fat line with the label ≪*Safety Transition*≫.

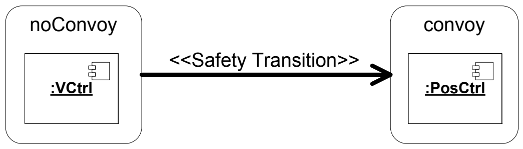


Figure 2: Safety Transition

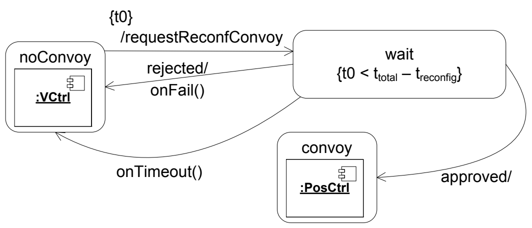


Figure 3: Safety Transition Semantics

The semantics of the Safety Transition are shown in Figure 3. We add the state *wait* between the states *noConvoy* and *convoy* connected by the Safety Transition. The reconfiguration request to the Risk Coordinator is triggered by the message *RequestReconfConvoy* of the transition $(noConvoy, wait)$. In case the Risk Coordinator answers with *approved* the reconfiguration is executed and the system switches to *convoy*. If the Risk Coordinator rejects the request, the system switches back to *noConvoy* and executes the side effect *onFail()*. onFail() is a method provided by the developer in order to react to the rejection. In our example the RailCab would inform the RailCab driving behind about the rejection and brake the convoy.

To guarantee a WCET for a Safety Transition, we add the clock $t0$ and a timing constraint: the invariant $\{t0 < t_{total} - t_{reconfig}\}$ for the state *wait*. t_{total} represents the time needed for the total Safety Transition. $t_{reconfig}$ names the time needed for the reconfiguration itself, namely the transition $(wait, convoy)$. Consequently, $t_{total} - t_{reconfig}$ is the time left for requesting the Risk Coordinator. The clock $t0$ is reset at transition $(noConvoy, wait)$. The invariant $\{t0 < t_{total} - t_{reconfig}\}$ guarantees that after requesting the Risk Coordinator there is still enough time left to execute the actual reonfiguration. If $t0$ exceeds the time limit, the system switches back to *noConvoy* with the side effect *onTimeout()*.

2.3 Allowing Required Reconfigurations

In some cases blocking transitions is not acceptable, e.g. reconfiguration in case of a component failure, or the request to the Risk Coordinator exceeds the time limits. Consequently, not all reconfiguring transitions can be Safety Transitions.

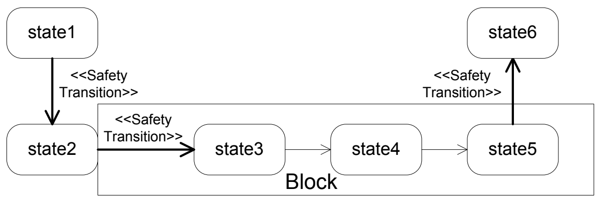


Figure 4: Safe Reconfiguration and Required Behavior

In order to still guarantee safe reconfigurations, we block transitions to configurations from which unsafe configurations are reachable via non blockable transitions. Figure 4 shows an examplary path containing non blockable transitions, which are drawn as thin lines. If the configuration in $state5$ was not allowed and we had to block transition $(state4, state5)$ as a consequence, we also would have to block transitions $(state2, state3)$ and $(state3, state4)$ (marked by a grey shadow). This check has to be applied to reconfigurations of the other system components as well, as a reconfiguration of one component can lead to a reconfiguration of another component.

Instead of sending a request to the Risk Coordinator when executing a non blockable transition, the component informs the risk coordinator of the executed reconfiguration. Since the non blockable transition must not be delayed, this information is sent immediately. Consequently, it is possible that the Risk Coordinator processes the update later than the reconfiguration takes place and doesn't know about the real system state. However, this is no threat to the safety as we already excluded unsafe configurations from the last blockable transition preceding the non blockable (cf. Figure 4).

2.4 Computing the Risk

The Risk Coordinator computes the risk of a configuration in order to decide whether the target configuration satisfies the maximum risk level or not. The risk is computed by multiplying the probability of the hazard in the target configuration with the damage of an accident that could result from the hazard. Currently, the hazard probabilities of the different system configurations are computed offline with the approach of [GT06] and stored in the Risk Coordinator. The damage is adapted continuously, e.g. when a RailCab is empty or contains hazardous materials.

2.5 Online Hazard Analysis

The approach presented so far only allows for the safe reconfiguration of a system of which all configurations are known before runtime as the hazard probabilities of all these configurations are pre-computed. For example, it is not possible to model infinitely long RailCab convoys as proposed in [THHO08]. In the following, we sketch an approach for computing the hazard probabilities during runtime.

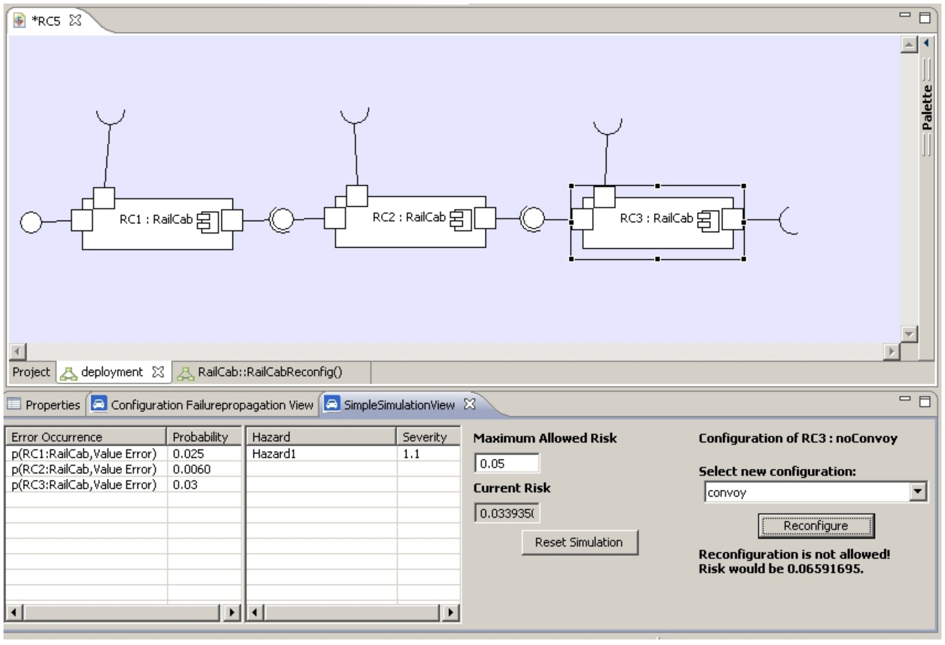
Since the current configuration of the total system is only known during runtime, the Risk Coordinator has to build the failure propagation model of the total system. The component's failure propagation model is stored in each component's Sub Coordinator. The Sub Coordinator transmits this failure propagation to the Risk Coordinator that computes the global failure model and performs a hazard analysis on the complete system. Once the hazard probabilities are known, the risk can be determined and the requested reconfiguration can be approved or rejected.

3. TOOL SUPPORT

The approach presented in this work has been implemented and embedded into several Fujaba4Eclipse plugins [THMvD08, BGH^{+}07]. These plugins support the modeling and analysis of safety-critical embedded systems with reconfiguration.

In this work we extended hybrid Reconfiguration Charts by Safety Transitions as presented in Section 2.2. We implemented a simulation interface for the Risk Coordinator component that supports checking whether configurations satisfy a given risk level. Figure 5 depicts the *Simulation View* of the plugin. The editor shows a component structure (configuration) consisting of three RailCabs. Below each system component is associated a hazard probability for a specific hazard. The total system is assigned a damage. Further, we can specify the maximum allowed risk for the total system and read the current risk. In order to simulate a reconfiguration, we select a component and choose a target configuration ("‘convoy"’) from the drop down menu. After pressing the Reconfigure button, the risk of the target configuration is computed and compared to the maximum allowed risk.

We are currently working on implementing the Risk Coordinator component as well as the online Hazard Analysis as sketched in Section 2.5 in order to fully integrate the presented approach into the Fujaba4Eclipse Real-Time Tool Suite.



Error Occurrence	Probability
p(RC1:RailCab,Value Error)	0.025
p(RC2:RailCab,Value Error)	0.0060
p(RC3:RailCab,Value Error)	0.03

Hazard	Severity
Hazard1	1.1

Maximum Allowed Risk
0.05
Current Risk
0.033935(

Configuration of RC3 : noConvoy
Select new configuration:
convoy

Reconfiguration is not allowed!
Risk would be 0.06591695.

Figure 5: Simulation Interface for the Risk Coordinator

4. CONCLUSION AND FUTURE WORK

We presented a risk analysis approach and its implementation in the Fujaba4Eclipse Real-Time Tool Suite which addresses self-* systems and dynamically changing damage values. The approach is twofold. First, the architecture is extended by a Risk Coordinator which checks whether a reconfiguration is safe w.r.t. the current damage. Second, the modelling language for the specification of the reconfiguration is extended by safety transitions which encapsulate the communication of each individual component with the Risk Coordinator.

The Risk Coordinator is a single point of failure in the current architectural approach. As we already consider self-* systems, we may add self-healing capabilities to the system as e.g. in [TG04] to fully operate even in case of failures.

It has to be noted that our approach requires quantitative data for both the hazard probability as well as the associated damage in order to compute the risk and to decide whether a configuration is allowed or not. Leveson [Lev95] says that quantitative data should be used with extreme care. Thus, we are currently investigating whether our approach can be extended to qualitative reasoning using probability and damage classes.

Our current approach does only block reconfigurations from a safe to an unsafe configuration. Due to changes to the damage during runtime a safe configuration may become unsafe. Therefore, we currently work on complementing our approach by forcing a reconfiguration from an unsafe to a safe configuration.

5. REFERENCES

[BGH^{+}07] Sven Burmester, Holger Giese, Stefan Henkler, Martin Hirsch, Matthias Tichy, Alfonso Gambuzza, Eckehard Müch, and Henner Vöcking. Tool support for developing advanced mechatronic systems: Integrating the fujaba real-time tool suite with camel-view. In *Proc. of the 29th International Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA*, pages 801–804. IEEE Computer Society Press, May 2007.

[GT06] Holger Giese and Matthias Tichy. Component-based hazard analysis: Optimal designs, product lines, and online-reconfiguration. In *Proc. of the 25th International Conference on Computer Safety, Security and Reliability (SAFECOMP), Gdansk, Poland*, Lecture Notes in Computer Science (LNCS), pages 156–169. Springer Verlag, September 2006.

[GTS04] Holger Giese, Matthias Tichy, and Daniela 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.

[Lap92] Jean Claude Laprie, editor. *Dependability : basic concepts and terminology in English, French, German, Italian and Japanese [IFIP WG 10.4, Dependable Computing and Fault Tolerance]*, volume 5 of *Dependable computing and fault tolerant systems*. Springer Verlag, Wien, 1992.

[Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[Sto96] Neil Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[TG04] Matthias Tichy and Holger Giese. A self-optimizing run-time architecture for configurable dependability of services. In Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting Dependable Systems II*, volume 3069 of *Lecture Notes in Computer Science (LNCS)*, pages 25–51. Springer Verlag, 2004.

[THHO08] Matthias Tichy, Stefan Henkler, Jörg Holtmann, and Simon Oberthür. Towards a Transformation Language for Component Structures. In *Postproc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany*, pages 27–39, 2008.

[THMvD08] Matthias Tichy, Stefan Henkler, Matthias Meyer, and Markus von Detten. Safety of component-based systems: Analysis and improvement using fujaba4eclipse. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE), Leipzig, Germany*, pages 1–2, May 2008.

[YA02] Sherif M. Yacoub and Hany H. Ammar. A methodology for architecture-level reliability risk analysis. *IEEE Trans. Softw. Eng.*, 28(6):529–547, 2002.

4 Tool Demonstrations

This section contains all background papers for the tool demonstration session.

Future Web Application Development with Fujaba

- visionary ideas, concepts and first results -

Christoph Eickhoff, Nina Geiger, Marcel Hahn, Ingo Witzky , Albert Zündorf
Kassel University, Software Engineering,
Department of Computer Science and Electrical Engineering,
Wilhelmshöher Allee 73,
34121 Kassel, Germany
[christoph.eickhoff | nina.geiger | hahn | ingo.witzky | zuendorf]@cs.uni-kassel.de
http://www.se.eecs.uni-kassel.de/

ABSTRACT

Nowadays, the trend of web applications getting more and more complex can barely be overseen. In some areas these web applications start to replace traditional desktop applications. The most important facts that lead to this trend are the possibility to use web applications from nearly every place in the world and that the end user does not need to install anything, nor to maintain the software or install updates etc.
While in the design and creation process of traditional software applications the model driven approach got its standing this is not yet the case in the development process of web applications. This paper introduces our ideas and concepts, as well as first results for web application development with model driven approaches. Of course, in this model driven web application development, Fujaba is our tool of choice.

1. INTRODUCTION

One of the strongest points of web applications is the reachability from every internet-connected place in the world and the loss of maintenace and software update tasks necessary to stay up to date. These advantages yield to web applications more and more replacing traditional desktop applications. The downside of this evolution is the growing complexity of web applications. This brings the need for different development methodologies and paves the way for traditional software engineering approaches adopted for the web domnain.

While for traditional applications model driven software development approaches have been introduced for years, this process has only started its way to the development of traditional web applications. There are a lot of research avtivities in this domain, for example the yearly workshop on Model-Driven-Web-Engineering [8]. The activities presented in conjunction with this workshop clearly focus on a more traditional web application. The evolution mentioned above leads to the need for development of AJaX (Advanced JavaScript and XML) web applications. These are more dynamic in behaviour and give us the opportunity to have business logic and model elements on client side. In former days, when web application meant static HTML pages and server side logic it was not necessary to really use any kind of software engineering methodology for the client side. Many of the older web applications were created by creative people who did web design and enriched the HTML content with small snippets of code written in JavaScript. Before the AJaX technology evolved it was barely necessary to have code on client side at all. The possibilities within the web and the growing complexity of web applications we have in our current times produces needs for web development in a more methodical way. The approach shown in this paper aims at the development of highly dynamic AJaX based applications. We will hold lots of business logic and an own data model on client side. This is done to distribute some of the calculation costs to the clients and thus unburden the server.

From the view of software engineers, complex software should be developed using the model driven approach. All the advantages of Model-Driven-Engineering (MDE) like shorter development cycles, better maintainability of the resulting code and better documentation of the whole project are commonly known in traditional software development. Surely the more complex web applications that are widely used today would benefit from all these advantages, too. As mentioned above, there have been scientific activities to built the domain of Model-Driven-Web-Engineering (MDWE) for a few years now. Focusing on the development of AJaX applications in a model driven way, we want to drive this community forward with our ideas.

Since from our point of view the Fujaba tool is the method of choice for model driven approaches, we now want to enable Fujaba for the web future. Beyond the ideas and code generation achievements introduced in [2] this paper talks about the overall ideas of enabling model driven web engineering with Fujaba. Since the design and implementation of graphical user interfaces has always been kind of special and often supported by special editors, we want to introduce our web engineering facilities from the very top (user interface) to the underlying business model.

Our paper is structured as follows:
Section 2 introduces our overall ideas for model driven web development and gives an overview of the architecture we achieve. It also gives a hint on the mapping of the GUI to the underlying business logic and model. After these ideas have been cleared, Section 3 introduces the current state of the development and focuses on our special graphical user interface (GUI) editor. Section 5 in the end summarizes our ideas and talks about the work to be done in near and far future.

2. FUJABA WEB DEVELOPMENT

As member of the Fujaba community, we are used to software engineering, using the Fujaba tool. For us, creating the model and developing the logical parts using activity diagrams is one of the best ways to develop software. Some of these steps have been brought to the development of web applications, already, cf. [2]. In this paper we presented the ablility to generate source code out of class diagrams that makes the model executable and usable inside the web browser. We introduced the special CodeGen2 [4] templates that replace the traditional collections used in Fujaba source code with collections from the java.lang package, there. We talked about the translation of the generated source code for use inside the web browser with the Google Web Toolkit (GWT) [11] Java to JavaScript cross compiler. We still rely on the GWT to deploy the generated Java code since this brings the advantage of being able to debug the web application using the GWT tools. In addition, GWT facilitates the usage of AJAX technologies, significantly.

Real model driven engineering with Fujaba goes further than just generating model code out of class diagrams. Of course we want to be able to design the logical parts of our web applications with Fujaba. This can be achieved for all parts that only affect the application model by generating GWT compliant code from the activity diagrams, too. We have to create templates similar to the ones for the class diagrams shown in [2] for the activities. This would mean to replace all occurences of Fujaba collections with the correspondent ones of java.lang within the activity templates. Doing this, the compilation with the GWT cross compiler should generate runnable JavaScript code for the web browser. But for complex web applications, we need more. We want to develop mechanisms to model client-server calls inside Fujaba. Here, we plan to use Statecharts alongside the just developed ideas from [7]. [7] describes the generic approach for the handling of property changes with Fujaba Statecharts. For client server communication within web applications this approach has to be slightly modified for the use with GWT Remote Procedure Calls. These calls are proceeded in an asynchronous way, which means the graphical user interface (GUI) will not block and wait for the answer of the server side. Because of that, the calling client method has to specify a so called callback object, which is registered and which will receive the answer from the server side. Since the approach introduced in [7] uses PropertyChange mechanisms, this has to be changed to enable the use of this callbacks for GWT like calls. One implementation that satisfies this callback approach has been developed at University of Kassel, Software Engineering Research Group to be used within the European project FAST (Fast and advanced Storyboard Tools)[10].

Another major point in generating web applications is the GUI the user will interact with. The creation of user interfaces has always been a case of trial and error and needed a lot of hand coding and testing until the user interface looked and worked as expected. There have been a load of GUI designer tools for every interface library one can imagine. Nevertheless, the only useable GUI designer working for GWT interfaces is commercial and thus it is not possible to use and extend it. Additionally, we want the GUI design to happen in concrete syntax, meaning the GUI designer tool itself will be a GWT application running inside a browser. To support our ideas for the generation of user interfaces and the binding of these interfaces to our Fujaba generated application model and logic, we develop an easy to use GUI designer for GWT widget libraries. These libraries are used in GWT to support the user interfaces and there are a lot of different ones, which we want to support using a modular plugin architecture for our designer. As mentioned above we will develop the user interface designer to handle concrete syntax, which means, the interface will be built out of GWT library components using a GWT application. This has the advantage, that the developer really sees how the resulting interface will look like, without having dummy representations of the resulting GUI objects. The interface is designed in a What-You-See-Is-What-You-Get (WYSIWYG) manner by dragging and dropping interface components from a sidebar to the main editing area. Figure 1 shows a mockup of the aspired designer and its functionality.

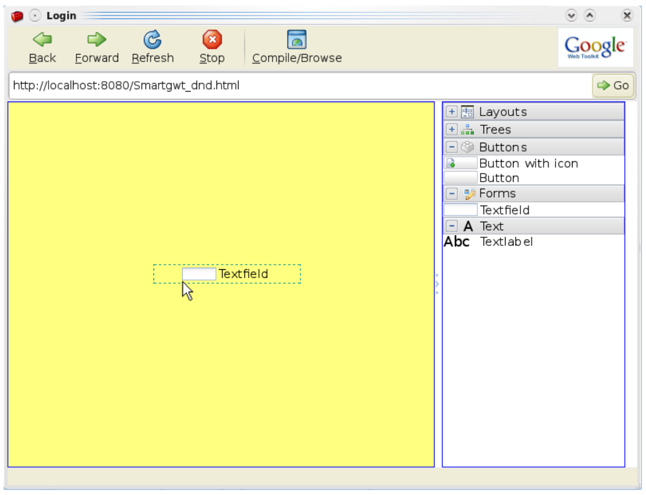


Figure 1: Mockup of the GUI designer.

To ease the overall web application development with Fujaba, the results of the GUI designer won't be Java source code, as it would be with other designers, but Fujaba activity diagrams building up the corresponding GUI. The challenge in this approach is the mapping between Fujaba meta model elements building up the activity diagram and the GUI elements building up the graphical user interface. Since the GUI designer will be written as GWT code and executed as JavaScript web application inside a web browser, we need to enable the editor to communicate with Fujaba to create the corresponding diagrams. This communication should also be established the other way round in case of changes occurring on the activity diagrams generated out of the editor. To achieve this communication, we will build up a wrapper structure for the user interface components. Since the GUI designer as an AJaX web application will run in browser mode, we will not be able to do the model mapping immediately, since it will be to heavyweight to run on client side. The mapping to the Fujaba meta model will take place on the server side of our designer, meaning inside Fujaba itself. We will have the complete wrapper structure on client side to give the editor a non-blocking user interaction, but we will need to replicate this wrapper data structure between client and server side. We developed an adaption for the CoObRA Framework [9] which enables the replication of models between a server and mutliple clients. This WebCoObRA approach is introduced in [1]. We will use this mechanism to synchrinize our wrapper structure between server and client. This enables us to do the mapping in real time, without having to save our GUI before we create the corresponding Fujaba diagram. As mentioned earlier, we will have to map the wrapper structure to Fujaba activity diagrams. To do this, we will use Triple Graph Grammars (TGG). For a more detailed view on TGGs see [5]. Using TGGs and the data replication between client and server side we are able to propagate all changes made within the GUI designer to the Fujaba activity diagrams and vice versa. Figure 2 gives a complete overview about the architecture we are focusing on.

Besides the mapping to Fujaba diagrams, the wrapper structure also enables us to have reflection on client side, which is not the case for the code of the widget libraries since there is no reflection in GWT. All necessary information about methods and attributes of the GUI components will be stored and retrieved inside the wrapper by as well the editor as the TGGs.

After the GUI has been created all the other web application code may be produced using Fujaba diagrams and methods. Using the adapted CodeGen2 templates, it is easy to generate executable web application code from Fujaba that may be debugged using the GWT Hosted Mode browser or deployed by compiling it to JavaScript using the GWT Compiler, cf. Figure 2.

3. FUJABA WYSIWYG

As mentioned above, currently there only exists one commercial WYSIWG-GUI-Editor as an Eclipse plugin [6]. This GUI designer supports the standard GWT widget library and has an extension for Ext JS [3]based widgets. Since it is commercial, there is no possibility to adopt this editor. Since we need to adopt the code generation aspects of the used editor to generate Fujaba diagram elements as well as we want to have a plugin architecture to support an infinite number of widget libraries, we currently focus on the development of our own GUI designer.

This application is implemented using the GWT and its SmartGWT widget library [12]. The first mockup of the designer can be seen in Figure 1. It consists of a main editing area and a sidebar containing all the GUI components / widgets. From the sidebar the components can be dragged to the editing area and dropped there. The sidebar entries contain references to the wrapper classes of every GUI component and thus ensure the right widget to be created on drop. We will try to develop templates to generate the wrapper classes automatically. These templates should then be adaptable to the different widget libraries one wants to embed into the editor. The development of these templates is future work.

For the handling of the GUI design, our editor will be enabled with property dialogs and handler objects, to ensure correct naming, parenting and size of the GUI component. For the assignment of EventHandlers to GUI components it will be possible to either use Fujaba or the property editors of the designer. In every case the implementation of the handler will take place inside Fujaba.

We also think of a special kind of adaption for the designer. There might be the case that a user wants to reuse one of his created GUIs. To do this, we want to enable the designer to store this GUI in a way that it can be added to the sidebar and used from there the next time the designer is startet. Imagine for example a Login screen, which consists of textfields for username and password as well as button for submission. If saved to the sidebar as *LoginWidget*, the next time a Login screen is needed, one only has to drag the *LoginWidget* from the sidebar and on drop all the containing elements will immediately be added to the editing area.

4. HOW TO INTEGRATE BUSINESS LOGIC

The integration of business logic to the web application will be done using the Fujaba tool. We will have all the methods and classes needed by the GUI available in Fujaba due to the mapping described above. The logic itself can be modelled using standard Fujaba mechanisms. Nevertheless, there will be the necessity to combine these logical operations with GUI elements and on top to define kind of screenflow for the web application. The concrete workflow for handling these requirements is not yet defined. We think of a fully integrated workflow for Fujaba which will enable the user to create web projects, define GUIs and combine these with the developed business logic. This will mean to develop new creation wizards, run the GUI designer from within Fujaba and of course, enable the assignment of logical elements to GUI and of EventHandlers to the GUI. For this assignment there will be the need to create new editors or diagrams which are not yet defined. The research in this area is future work and will be published later.

The business logic iteself, be it EventHandlers or more complex operations on the model will be created using Fujabas activity diagrams. For all operations that will need to run on the client side of the web application the codestyle has to be set to “GWT”. Fujaba will do the rest for you.

5. SUMMARY AND FUTURE WORK

This paper showed our ideas on enabling the Fujaba tool for the creation of complex web applications. We talked about the steps currently carried out. We tried to motivate the advantages and needs of model driven development for web applications and talked about the creation of user interfaces in more depth. We introduced the basic function of our GUI designer mockup and indicated the possible workflow we want to achieve.

Since lots of our ideas still have the status of visions, we have a lot of work to do in the future. Our first achievements of GUI creation have to be carried on. The automatic generation of the wrapper classes needed will be the next topic to focus, as well as the mapping of these to the Fujaba meta model. We also will have to enhance the usability of our GUI designer, to make it as intuitive as possible. Another important item on our todo list will be the modularization of our code, to give it the plugin architecture mentioned. We have to focus on a simple API to extend the designer with further widget libraries. This extension will have to comprise wrapper classes as well as TGG rules for the mapping. To attract widget library owners to extend our designer with their libraries, we have to ease this step as much as possible.

It will be necessary to extend Fujaba and its wizards for the creation of web projects and Web UIs. These wizards shall create the basic structure needed for a web application project intended to run with GWT and WebCoObRA as well as some necessary files, like the html startpage and xml files needed by GWT.

In the end the saving of widgets to the sidebar has to be implemented and the corresponding wrapper classes have to be generated from the ones contained in the widget that was saved.

6. REFERENCES

[1] N. Aschenbrenner, J. Dreyer, M. Hahn, R. Jubeh, C. Schneider, and A. Zündorf. Building Distributed Web Applications based on Model Versioning with CoObRA: an Experience Report. In *Proc. 2009 Intl. Workshop on Comparison and Versioning of Software Models*, pages 19–24. ACM, May 2009.

[2] N. Aschenbrenner, J. Dreyer, R. Jubeh, and A. Zündorf. Fujaba goes web 2.0. In U. Aßman, J. Johannes, and A. Zündorf, editors, *6th International Fujaba Days*, pages 10–14, Dresden, Germany, 2008.

[3] Ext JS. http://www.extjs.com/, 2009.

[4] L. Geiger, C. Schneider, and C. Reckord. Template- and modelbased code generation for MDA-Tools. In *3rd International Fujaba Days*, Paderborn, Germany, September 2005.

[5] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy*, volume 4199 of *Lecture Notes in Computer Science (LNCS)*, pages 543–557. Springer Verlag, 10 2006.

[6] Instantiations - GWT Designer. http://www.instantiations.com/gwtdesigner/, 2009.

[7] R. Jubeh and A. Zündorf. Propertychange events meet fujaba statecharts. submitted to 7th International Fujaba Days, 2009.

[8] Yearly held workshop on Model-Driven Web Engineering. http://mdwe2009.pst.ifi.lmu.de/, 2009.

[9] C. Schneider. *CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten.* PhD thesis, 2007.

[10] FAST. http://fast.morfeo-project.eu/, 2009.

[11] The Google Web Toolkit. http://code.google.com/webtoolkit, 2009.

[12] SmartGWT Widget Library. http://code.google.com/p/smartgwt/, 2009.

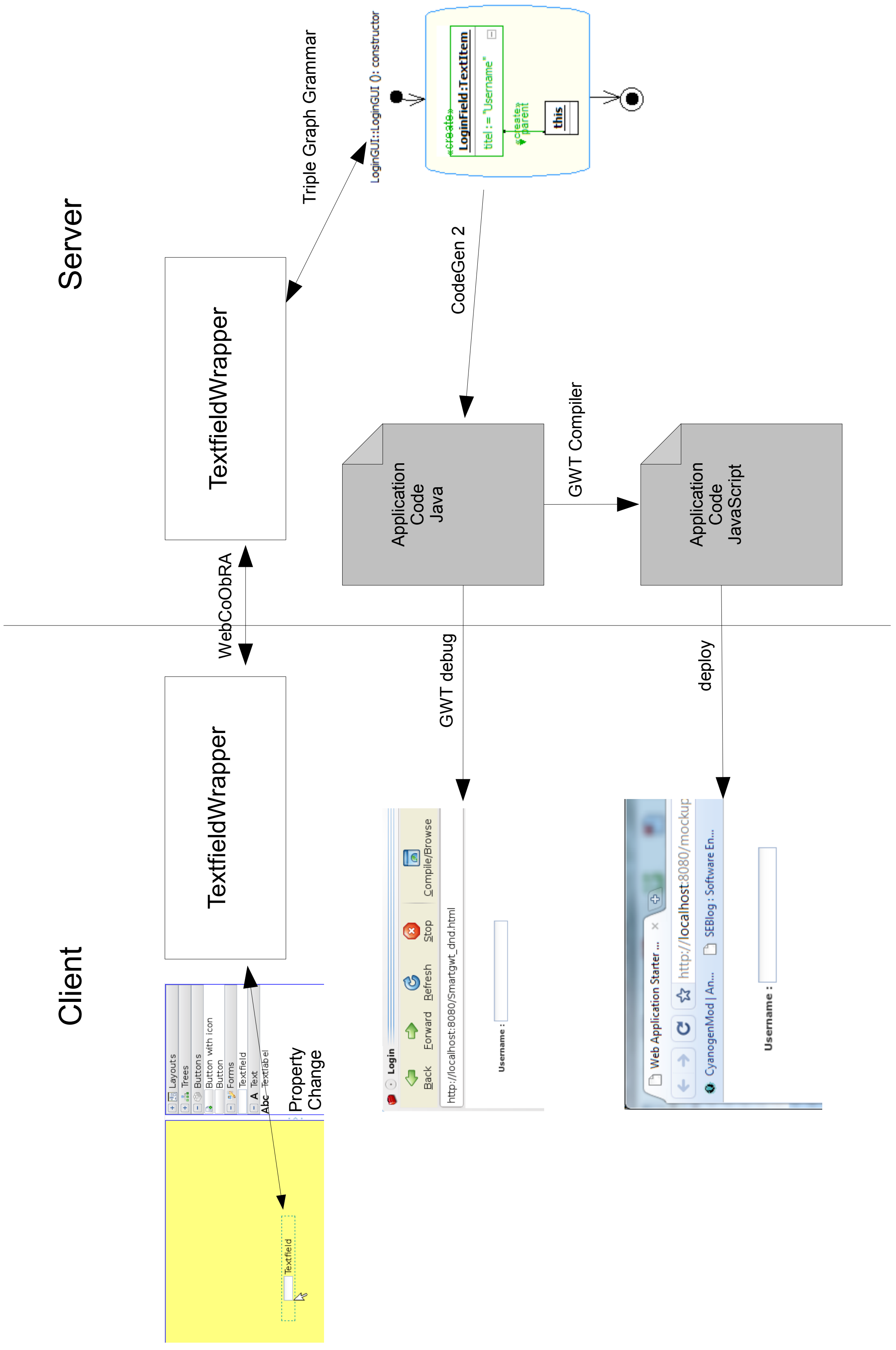


Figure 2: Overview of Fujaba Web Development components and communication flow.

NT2OD

From natural text to object diagram

Jörn Dreyer, Albert Zündorf
Kassel University, Software Engineering,
Department of Computer Science and Electrical Engineering,
Wilhelmshöher Allee 129b
34121 Kassel, Germany
[jdr | zuendorf]@cs.uni-kassel.de
http://www.se.eecs.uni-kassel.de/index.php?joern

ABSTRACT

This paper documents the initial results of an attempt to create tool support for the "objects first" development process. The idea is to apply natural language processing (NLP) and ontology learning techniques to textual use case descriptions and derive an initial object diagram that can be refined by the software developer. The prototype is a proof of concept for simple sentences and will be further improved.

1. INTRODUCTION

The software development process of Fujaba teaches "objects first" as described in [2]. To find these objects use case descriptions are scanned for striking parts of speech:

- nouns indicate objects,
- verbs indicate relations or methods, and
- adjectives indicate attributes.

The designer has to decide about an actual mapping and then creates an object diagram representing the use case.

The NT2OD prototype[1] tries to automate this manual task for the english language by applying NLP algorithms. The global workflow consists of three steps:

1. Create a parse tree for the sentence.
2. Identify objects and relations.
3. Create the Fujaba object diagram.

A lot of work has already been done by linguists so that NT2OD can reuse existing technologies.

[1]availiable as a Fujaba plugin:
https://gforge.cs.uni-kassel.de/projects/nt2od/

2. RELATED WORK

In NLP statistical parsers currently dominate the approaches to the first step and have been explored thorougly by linguists. The availiable implementations for Java include, but are not limited to the following projects:

2.1 OpenNLP Tools

The OpenNLP Tools are developed as a sourceforge project[2] and use the maximum entropy framework implemented by the opennlp.maxent package[3] based on [7]. The english models have been trained with the Penn Treebank corpus. However, a statement on precision or recall could not be found. It can do POS tagging, named-entity detection, coreference resolution and anapher resolution.

2.2 The Stanford parser

Based on [5] and [4] the "Stanford Natural Language Processing Group" developed the Stanford parser. The lexicalized probabilistic parser has been trained on the Penn Treebank corpus and yields a precision of 86.6% and a recall of 86.8%. It provides POS tagging and the generation of parse trees.

2.3 GATE

The "General Architecture for Text Engineering" described in [1] has been developed by the Natural Language Processing Research Group[4] at the University of Sheffield[5]. The default set of processing resources is called ANNIE[6] and contains a tokenizer, sentence splitter, gazetter, transducer, POS tagger, (pro-)nominal coreferencer and ortographical coreference component for the english language. A verb phrase chunker can be added from the "Tools" section of the CREOLE plugins and a noun phrase chunker based on [6] is also availiable.

The POS tagger with a tagging accuracy of around 97% has been trained on a large corpus taken from the Wall Street Journal using a modified version of the Brill tagging algorithm described in [3].

[2]http://Fopennlp.sourceforge.net/
[3]http://maxent.sourceforge.net
[4]http://nlp.shef.ac.uk/
[5]http://www.shef.ac.uk/
[6]A Nearly New IE system

Finally, the SUPPLE plugin[7] provides a parse tree like the other parsers.

2.4 Text2Onto

Text2Onto[8] is a framework for ontology learning from text that is based on GATE and WordNet[9]. However, it is designed to customize the ontology learning algorithm and toolchain when working with large corpora whereas NT2OD is focused on the task of interactively mapping single sentences and short texts to object diagrams. Still, NT2ODs main concepts have been greatly inspired by Text2Onto.

3. AUTOMATING "OBJECTS FIRST"

The first step in NT2OD resembles the human process of identifying Parts of Speech (POS). However, the process of identifying concepts can be improved by also identifying the hierarchical structure of the sentence – the "parse tree".

3.1 Creating parse trees

While several projects for natural language processing are availiable the prototype initially has been developed with OpenNLP[10] for its speed and ease of integration. The actual parsing process is best described by an example sentence:

```
Alice and Bob are playing Ludo.
```

Simple part of speech tagging attaches short tokens to the words, identifying its class[11]:

```
Alice/NNP[12] and/CC[13] Bob/NNP are/NNP
playing/VBG[14] Ludo/NNP ./.
```

As you can see the tagger made a mistake and assigned "proper noun" to the verb `are`. Switching OpenNLP to parse trees not only reveals the the sentence structure, it also improves precision:

```
(TOP (S[15] (NP[16] (NNP Alice) (CC and)
(NNP Bob)) (VP[17] (VBP[18] are)
(VP (VBG playing) (NP (NNP Ludo.))))))
```

Meanwhile, a parser interface has been introduced to the prototype, enabling users to switch between OpenNLP, Stanford and soon GATE. A small adapter is needed as parser output differs marginally, e.g. sentence delimiter bracketing.

[7]http://gate.ac.uk/sale/tao/#x1-2280009.12
[8]http://ontoware.org/projects/text2onto/
[9]http://wordnet.princeton.edu/
[10]http://opennlp.sourceforge.net/
[11]http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html
[12]NNP: proper noun, singular
[13]CC: coordinating conjunction
[14]VBG: verb, gerund or present participle
[15]S: sentence
[16]NP: noun phrase
[17]VB: verb phrase
[18]VBP: verb, non-3rd person singular present

3.2 Extracting concepts

The parse tree could be used to directly create the fujaba model, however creating an intermediate layer eases the representation of alternatives (should a verb be mapped to a relation or a method). Consisting of concepts, instances and relations between them, it also allows an easy reidentification of conceps by its properties which gains importance when merging them later in the process. First, the prototype creates an object world that mimics the parse tree and then tries to identify subject, predicate and object for a simple grammar, as shown in figure 1.

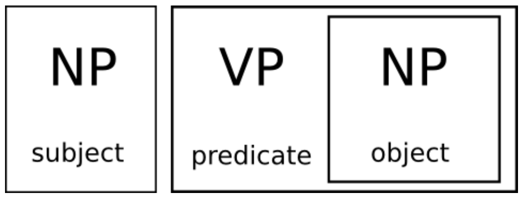


Figure 1: Simple sentence structure

All nouns in the first noun phrase become subjects, the last verb becomes the predicate and the nouns of the verbs noun phrase become objects (in the sense of sentence grammar). Any proper nouns are currently used as the object name. Other nouns become class names. However, the explicit assignment of an instance to a class in the form of the verb *"to be"* only creates a new inheritance in order not to overwrite a previously assigned class, preventing loss of information. This open world assumption creates a lot of classes, but it leaves the choice of which classes to merge to the user. Figure 2 shows the class diagram for our example sentence.

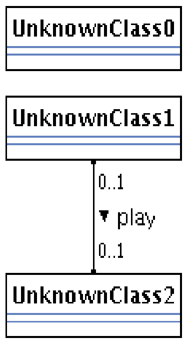
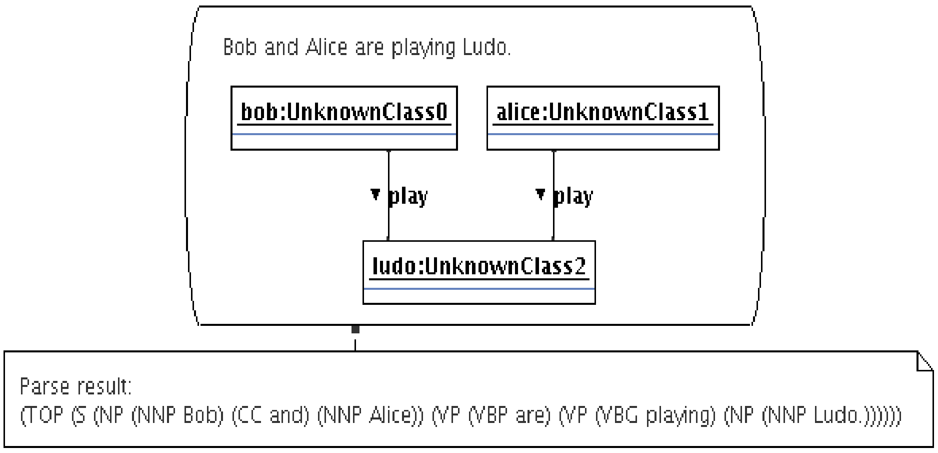


Figure 2: Class diagramm after parsing

During the development of the prototype the assumption was made to map predicates to relations instead of methods, as this better fits the creation of object diagrams. Furthermore, quantities like `four players`, as well as a few colors (those available in `java.awt.Color`) are taken into account when creating objects and determinig the multiplicity of an association in the class diagram.

3.3 Visualizing concepts

The third step to achieve "objects first" tool support is the visualization of the intermediate layer in Fujaba. When



Parse result:
(TOP (S (NP (NNP Bob) (CC and) (NNP Alice)) (VP (VBP are) (VP (VBG playing) (NP (NNP Ludo.))))))

Figure 4: Object diagram after parsing

transforming it into a Fujaba model (mainly `UMLObject` and `UMLClass`) the prototype creates a new name *"Unnamed-Class"* for every unnamed class and appends a serial number. Anonymous objects derive their name from their class and also append a serial number. In order to visualize links (`UMLLink`) between objects Fujaba also requires the corresponding association (`UMLAssoc`) to be set. Currently the parsing process is started by choosing the new "Parse Annotation" action in the popup menu of a storyboard as seen in figure 3. Figure 4 shows the object diagram and parse result for the example sentence.

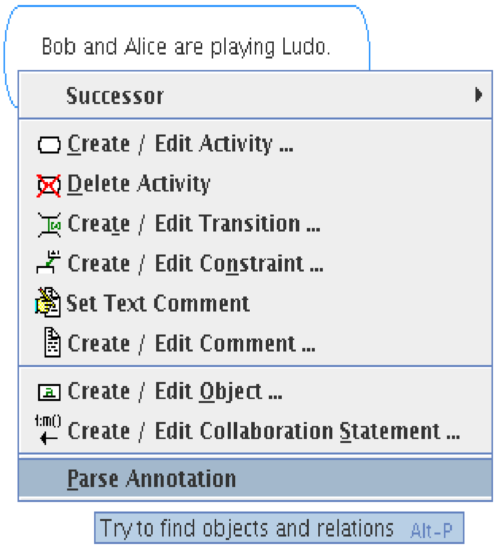

Bob and Alice are playing Ludo.
Successor
Create / Edit Activity ...
Delete Activity
Create / Edit Transition ...
Create / Edit Constraint ...
Set Text Comment
Create / Edit Comment ...
Create / Edit Object ...
Create / Edit Collaboration Statement ...
Parse Annotation
Try to find objects and relations Alt-P

Figure 3: Popup menu

4. CONCLUSIONS

Although the process of identifying objects and relations is heavily influenced by the sentence structure understood by NT2OD, the current implementation of the prototype has proven to be a working solution for simple sentence structures. In the future we plan to switch from a hard coded model and concept extraction to Fujaba activity diagrams to visualize the transformation between parse tree and object diagram. This will allow us to widen the range of sentence structures and make NT2OD more flexible. Furthermore, the inclusion of GATE will improve the precision of parse trees and in consequence the resulting object diagrams.

5. REFERENCES

[1] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, 2002.

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

[3] M. Hepple. Independence and commitment: Assumptions for rapid training and execution of rule-based pos taggers. In *Proceedings of the 38 th Annual Meeting of the Association for Computational Linguistics*, Hong Kong, October 2000.

[4] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Annual Meeting of the Association for Computational Linguistics*, volume 41, pages 423–430, 2003.

[5] D. Klein and C. D. Manning. Fast exact inference with a factored model for natural language parsing. In *Advances in Neural Information Processing Systems*, volume 15. MIT Press, 2003.

[6] L. A. Ramshaw and M. P. Marcus. Text chunking using transformation-based learning. In *Proceedings of the Third ACL Workshop on Very Large Corpora*, 1995.

[7] A. Ratnaparkhi. *Maximum Entropy Models for Natural Language Ambiguity Resolution.* PhD thesis, University of Pennsylvania, Philadelphia, PA, 1998.

Supporting Modeling in the Large in Fujaba

Thomas Buchmann
Angewandte Informatik 1
Universität Bayreuth
D-95440 Bayreuth
thomas.buchmann@uni-bayreuth.de

Alexander Dotor
Angewandte Informatik 1
Universität Bayreuth
D-95440 Bayreuth
alexander.dotor@uni-bayreuth.de

Martin Klinke
Xavo AG
Enterprise IT Solutions
D-95444 Bayreuth
martin.klinke@xavo.com

ABSTRACT

Model-driven software development intends to reduce development effort by generating code from high-level models. However, models for non-trivial problems are still large and require sophisticated support for modeling in the large. Experiences from our current project, dedicated to a model-driven modular software configuration management (SCM) system, confirm this claim. This paper presents a stand-alone package diagram editor that has been developed using Fujaba and discusses its integration into the Fujaba tool suite.

Keywords

package, import, package diagram, model-driven development, modeling in the large, validation

1. INTRODUCTION

In object-oriented modeling, modeling in the large is an area which has not attracted sufficient attention, so far. The focus in object-oriented modeling tools is clearly on the level of class diagrams. When it comes to modeling the architecture of a system, particularly the dependencies between certain modules, package diagrams play a crucial role. The experiences gained in our project, dedicated to the development of a model-driven product line of software configuration management systems, showed us that managing the dependencies between packages is mandatory for the overall success of the project [4].

UML 2.0 offers concepts for structuring large models [6, 1]: A model may be structured into hierarchically organized *packages*. Each model element is owned by exactly one package. *Private elements* are not accessible from other packages, while *public elements* are visible. Each package defines a *namespace* in which the names of declared model elements have to be unique. Public model elements from other packages may always be referenced through their fully qualified names. A model element from an enclosing package may be referenced without qualification, unless it is hidden by an inner declaration.

Apart from nesting, UML 2.0 introduces the following relationships between packages: *Imports* merely serve to extend the set of those elements which may be referenced without qualification. UML 2.0 distinguishes between *public* and *private* imports, which are denoted by the stereotypes <<import>> and <<access>>, respectively. A private import makes the imported elements visible only in the importing package, while a public import simultaneously adds those elements to its exported namespace. Public imports are transitive; this property does not hold for private imports. UML 2.0 also offers *package merges*, which will not be discussed in this paper.

In this paper, we present the *MODPL package diagram editor*, which is a first step of supporting UML 2.0 package diagrams, either as a stand-alone editor or within the Fujaba tool suite. Furthermore, it can be used to validate Fujaba models against a pre-defined package diagram.

2. PACKAGE DIAGRAM META-MODEL AND EDITOR

Our *MODPL package diagram editor* has the capability of creating and editing UML compliant package diagrams. Basically it is **independent** of Fujaba or Ecore. But its meta-model is designed to provide an interface for third party tools - either to **import** an already existing package structure, or to **validate** visibility constraints.

The **package diagram meta-model** itself is a subset of the meta-model described in the UML specification [6]. It contains the following elements:

- Packages
- Classes
- Package imports (Private and Public)
- Element imports (Private and Public)

Currently package merges are omitted in the meta-model. Take note that a package diagram model contains **only** information about the **visibility** and no other relationships between packages or classes.

Additionally, the meta-model provides an **interface** to *create*, *delete* and *validate* model elements which is solely based on **fully qualified names**. This way, the interface is independent of the 3rd-party meta-models the package diagram editor has to work with: *Ecore*, *Fujaba* or any other UML-based meta-model can be integrated – identity of elements is provided by the fully qualified names.

The **editor** was developed in a pure model-driven way. In particular, *Fujaba* was used to describe the meta-model – including an interface and behavior to conduct visibility checks

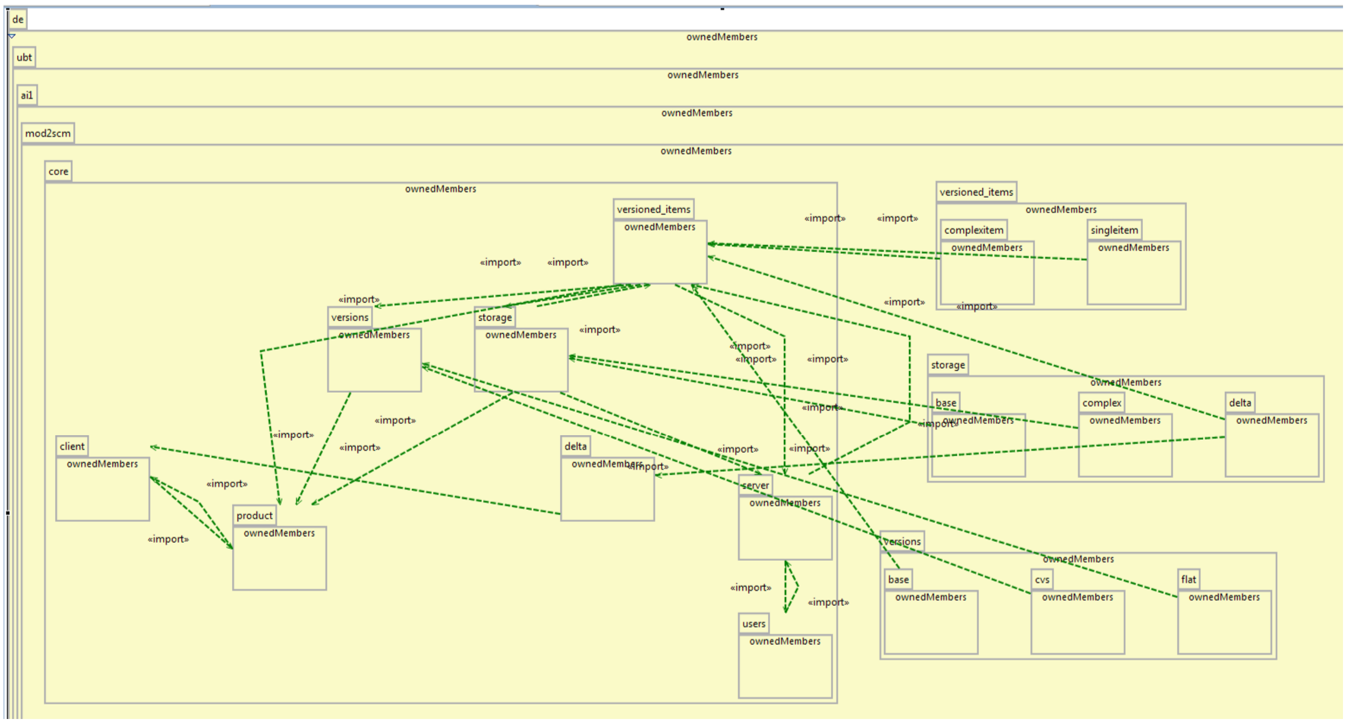


Figure 1: The package diagram of the MOD2-SCM project created with the package diagram editor.

– and exported as *Ecore* model, using Fujaba's code generation engine for the *Eclipse Modeling Framework* (EMF) [5]. Then the *Eclipse Graphical Modeling Framework* (GMF) was used to generate the graphical editor, based on our previous experiences [3]. Figure 1 shows a package diagram of our *MOD2-SCM* project.

3. INTEGRATION INTO EMF AND FUJABA

The ultimate goal of the MODPL package diagram editor is to become integrated with other tools that are based on models containing the concept of package diagrams, but lack a package diagram editor itself: like *Fujaba*. In the following sections we describe the various steps we have undertaken, so far, towards the goal of a seamless integration with Fujaba and EMF.

3.1 Step 1: Deriving package diagrams from Ecore

It is very easy to generate package diagrams on top of an already existing Ecore model definition (see the left dependency in Figure 2). The **import mechanism** reads the Ecore file and creates a corresponding package diagram file containing all packages and classifiers (Classes and Datatypes) from the imported model. Additionally, package or element imports can be deducted from the given Ecore model. After that, the graphical representation of the package diagram can be initialized.

3.2 Step 2: Validating Ecore models against package diagrams

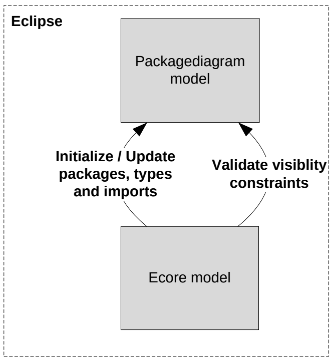

Eclipse
Packagediagram model
Initialize / Update packages, types and imports
Validate visiblity constraints
Ecore model

Figure 2: Step 1&2 – Dependencies between Ecore and package diagram models.

The validation checks if for each model element all referenced model elements are visible (see the right dependency in Figure 2). The validation process works as follows: The **source** and **target** element are specified by their fully qualified name, and a request is made – using our meta-model's interface – if the target is visible to the source. This distinction is necessary as visibility is not symmetric – if element A is visible to element B they are not necessarily vice versa.

To keep the package diagram small and readable, we require only the packages to be contained in the package diagram. Classes can be omitted and should be, as long as they are no target of an import. Therefore, four possible results can occur when source and target are validated.

1. **both** elements are contained in the package diagram
2. only the **target** is part of the package diagram
3. only the **source** is part of the package diagram
4. **both** source and target are **not** contained in the package diagram

If an element is not contained in the package diagram, the containing namespace – in most cases a package if no inner class is checked – is used for the validation, instead of the element itself. The fully qualified name of the containing namespace can be computed by simply **truncating** the element's fully qualified name before the **last dot**.

If a violation occurs, we have two possible **origins**:

1. The validated model uses an **non-visible type** and has to be fixed.
2. The package diagram **misses** one or more **imports** which have to be added.

3.2.1 *Ecore validation plugin*

To validate an Ecore model, we chose to use the *EMF Validation Framework* to trigger the constraint checking and the *Object Constraint Language* (OCL) to specify a constraint on each Ecore element. The OCL statement simply **delegates** source and target name to the appropriate package diagram model method – and does not specify the constraint itself. This allows us to specify the constraint checks in Fujaba without loosing the capability of the Eclipse Validation framework to mark violating model elements visually and to link them with the corresponding error message. During the validation process, the containment tree of the Ecore model is traversed, and the following meta-types of Ecore elements are validated:

- EReference
- EOperation
- EParameter
- EAttribute
- EClass

Source and target are determined as follows: The source is the owner of the Ecore element, whereas the target is its type (i.e. their fully qualified names).

3.3 Step 3: Deriving package diagrams from Fujaba

To create a package diagram for a Fujaba model, it is exported to EMF first, and the generated Ecore model used to initialize the package diagram. This way Ecore acts as **intermediate model** between Fujaba and the MODPL package diagram model (as depicted in Figure 3).

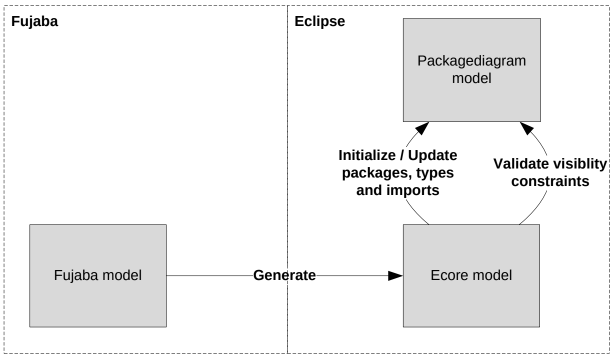

Fujaba
Eclipse
Packagediagram model
Initialize / Update packages, types and imports
Validate visiblity constraints
Fujaba model
Generate
Ecore model

Figure 3: Step 3 – Using Ecore as intermediate model.

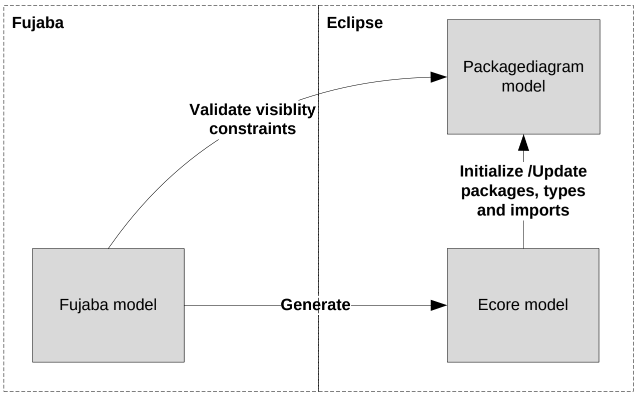

Fujaba
Eclipse
Packagediagram model
Validate visiblity constraints
Initialize /Update packages, types and imports
Fujaba model
Generate
Ecore model

Figure 4: Step 4 – Validating Fujaba models directly.

3.4 Step 4: Validating Fujaba models against package diagrams

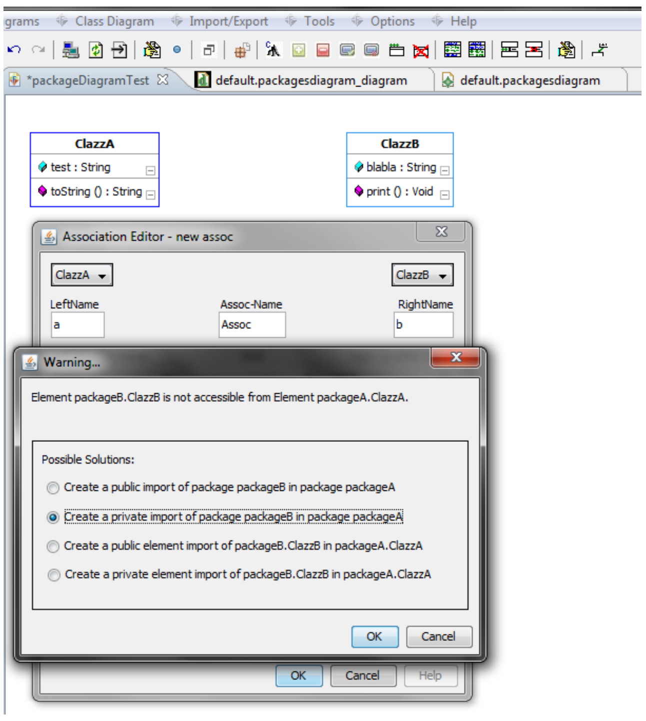
By passing the fully qualified names from Fujaba to the corresponding package diagram model, it is possible to check the visibility between two Fujaba elements. Figure 4 depicts the dependencies between the three models: Fujaba, Ecore and package diagram model.

3.4.1 *Fujaba validation plugin*

A connector has been implemented as a Fujaba plugin, which combines the package diagram editor presented in this paper and *Fujaba4Eclipse* with the *SwingUI*. In the Fujaba tool suite, an existing package diagram model (created with our package diagram editor) is loaded during the editing process. After the model has been loaded, the visibility of model elements is checked when

1. an **association** is created.
2. **attributes** are added.
3. **return types** of methods are selected.
4. **parameters** of methods are specified.
5. the type of an **object** is selected in a story diagram.

The plugin is even capable of updating the package diagram in case a visibility constraint is violated.



Element packageB.ClazzB is not accessible from Element packageA.ClazzA.

Possible Solutions:

- ( ) Create a public import of package packageB in package packageA
- [✓ Create a private import of package packageB in package packageA]
- ( ) Create a public element import of packageB.ClazzB in packageA.ClazzA
- ( ) Create a private element import of packageB.ClazzB in packageA.ClazzA

Figure 5: A simple example for the integration of the package diagram editor in Fujaba.

3.5 Step 5: Updating the package diagram from Fujaba

In case a visibility constraint is violated in Fujaba (i.e. a selected element is not visible), the **user** can decide whether he wants to **cancel** the operation in order not to violate the import dependencies specified in the package diagram, or to **add** the appropriate kind of **import** automatically to the package diagram (public or private package import or public or private element import). A simple example is given in Fig. 5: The example consists of two classes, `ClazzA` and `ClazzB`, both defined in package `packageA` and `packageB`, respectively. The corresponding package diagram does not define any imports between the two packages. Now an association is added between `ClazzA` and `ClazzB`. Figure 6 shows the package diagram after the association was added and the user selected the kind of import that should be added automatically to the package diagram (in this case a private package import was used).

The current dependencies between Fujaba and the package diagram model are depicted in Figure 7. Please note that the package diagram editor can be used on existing Fujaba models - but these have be imported using the editor's Ecore import mechanism. Therefore, the Fujaba model has (still) to be exported to Ecore using the EMF code generation. Once the import has been completed (and the package diagram has been associated with the Fujaba model), the validation plugin is used to keep the package diagram up-to-date.

4. RELATED WORK

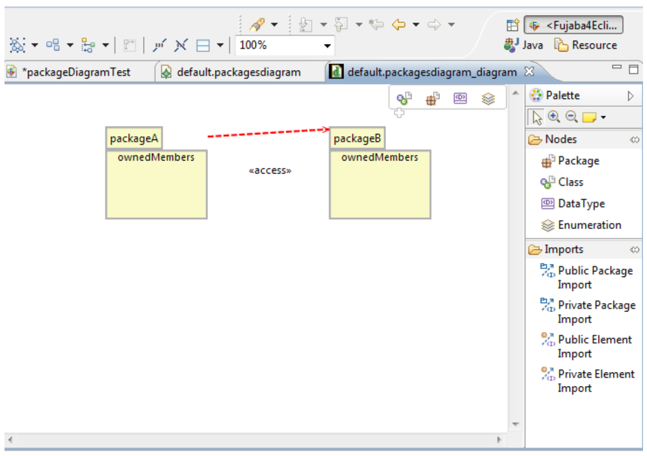


Figure 6: The package diagram after the association was added in Fujaba.

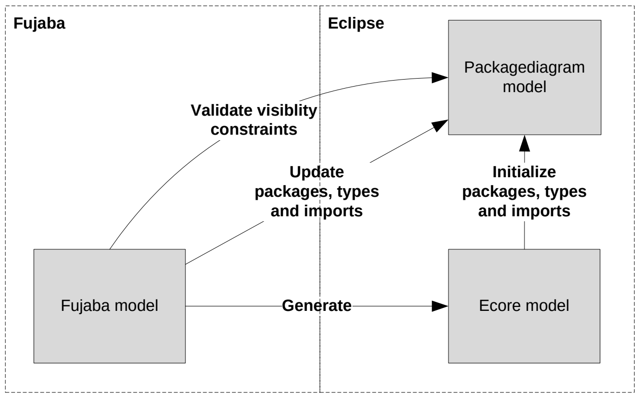

Fujaba
Eclipse
Packagediagram model
Validate visiblity constraints
Update packages, types and imports
Initialize packages, types and imports
Fujaba model
Generate
Ecore model

Figure 7: Step 5 – Current state: Updating package diagram models from Fujaba.

Fujaba itself provides only a package tree classes can be assigned to, but no package diagrams. *MOFLON* [2], which has been built on top of Fujaba, offers package diagram capabilities based on the MOF 2.0 meta-model. The MOFLON editor filters the elements based on the visibility constraints. However, it supports, currently, neither the update of the package diagram if a visibility constraint is violated nor the integration of other meta-models. *UML2Tools* for Eclipse provides a mature package diagram editor for UML2 models. These models can be used to generated EMF models so it is possible to use these tools for large EMF models. However, neither does the graphical UML2 editor support package imports (although the metamodel does) nor does the metamodel validation check the visibility constraints. The latter is common behavior for UML package diagram editors: If an element is not visible the fully qualified name is used to access the element – without any warning or error.

5. FUTURE WORK

The next step is the complete integration of our package diagram editor into the Fujaba tool suite (as depicted in Figure 8). The validation plugin has to traverse the Fujaba model and to **create** a corresponding **package diagram** model.

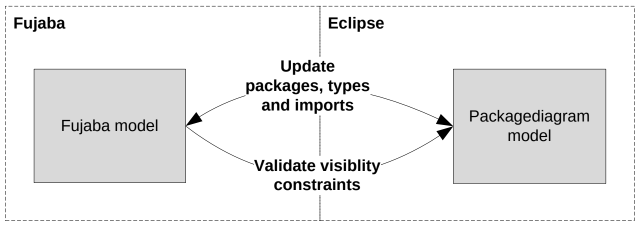

Fujaba
Eclipse
Fujaba model
Update packages, types and imports
Validate visiblity constraints
Packagediagram model

Figure 8: Step 6 – Future work: Removing the intermediate Ecore model.

Further extensions affect the incorporation of the visibility information into the **user interface**: It can be used to limit the referencing of model elements only where they are visible. For example, the class diagram editor can only show elements which are visible from the package the class diagram belongs to. Furthermore, a slight change to the Fujaba meta-model is recommended: not only model elements such as classes and associations, but also class and story diagrams should be assigned uniquely to one package. The package diagram editor itself can also be extended in various ways. For example, besides the view with nested packages, different visualizations could be chosen to provide a more flexible view of packages and their interdependencies to the user.

6. CONCLUSION

In this paper, we presented a package diagram editor developed with Fujaba and GMF. It allows the user to validate Fujaba and Ecore models against package diagrams with import dependencies – and keep them up-to-date. The package diagram can be either created from scratch, or it can be initialized from already existing Ecore models. In the last section, we discussed possible integration points of the package diagram editor into the Fujaba tool suite.

7. ACKNOWLEDGMENTS

We would like to thank our supervisor, Prof. Dr. Bernhard Westfechtel, for the invaluable input during the development of the package diagram editor and his feedback as a reader of this paper.

8. REFERENCES

[1] *OMG Unified Modeling Language (OMG UML), Infrastructure, V 2.1.2*, Nov. 2007. http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF.

[2] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A standard-compliant metamodeling framework with graph transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume LNCS 4066, pages 361–375, Genova, Italy, October 2006 2006.

[3] T. Buchmann, A. Dotor, and B. Westfechtel. Model-driven development of graphical tools - fujaba meets GMF. In J. Filipe, M. Helfert, and B. Shishkov, editors, *Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT 2007)*, pages 425–430, Barcelona, Spain, July 2007. INSTICC Press, Setubal, Portugal.

[4] T. Buchmann, A. Dotor, and B. Westfechtel. Experiences with modeling in the large with fujaba. In U. Assmann, J. Johannes, and A. Zündorf, editors, *Proceedings of the 6th International Fujaba Days*. University of Dresden, University of Dresden, 2008.

[5] L. Geiger, T. Buchmann, and A. Dotor. EMF code generation with fujaba. In L. Geiger, H. Giese, and A. Zündorf, editors, *Proceedings of the 5th International Fujaba Days*, 2007.

[6] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure*. OMG, November 2007. Version 2.1.2.

$FRiTS^{Cab}$: Fujaba Re-Engineering Tool Suite for Mechatronic Systems[*]

Stefan Henkler, Moritz Breit, Christopher Brink, Markus Böger, Christian Brenner, Kathrin Bröker, Uwe Pohlmann, Manel Richtermeier, Julian Suck, Oleg Travkin, Claudia Priesterjahn
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[shenkler,mbreit,cbrink,markusb,cbr,kathyb,upohl,mane02,jsuck,oleg82,cpr]@uni-paderborn.de

ABSTRACT

Mechatronic systems use their software to enable enhanced functionalities. Due to the complexity of these systems model-driven engineering of the software has become the means to construct reliable software. As safety is of paramount importance for these systems, legacy components, which have shown their quality in practice, are often reused. Therefore, the re-engineering of legacy components has to taken safety requirements into account. We present an approach for the re-engineering of mechatronic systems which focuses especially on distributed real-time and safety requirements and an integration into a model-driven engineering approach.

1. INTRODUCTION

Mechatronic systems, like automotive systems and aerospace systems, are usually networks of mechatronics components. Software is used to enable communication and thus to exploit knowledge of other components to enhance the functionality and to adapt the behavior of a single component when beneficial. Adapting the behavior might require complex reconfigurations of controllers in the form of mode management and control algorithms under hard realtime constraints. Due to the complex nature of networked mechatronic systems and their usually safety-critical operations, e. g. lives may be at risks in case of failure, model-driven engineering of the software has become the means to construct reliable software.

As safety is of paramount importance for mechatronic systems, legacy components, which have shown their quality in practice, are often reused. Therefore, the integration of legacy components into a model-driven engineering approach is obvious [Bal00]. The integration postulates that an adequate model of the legacy system is available. The automatic extraction of the models of a legacy component based on the existing executable code and its defined interfaces is the topic of this paper.

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

The overwhelming complexity of the interaction of distributed real-time components usually excludes that testing alone can provide the required coverage when integrating a legacy component. Thus formal verification techniques seem to be a valuable alternative. However, the required verification of the resulting system often becomes intractable as no abstract model of the reused components is available, which can serve the verification purpose.

As the origin of a legacy component could be different from case to case, the known information of the legacy component differs, too. The level of detail of the information is essential for the required techniques and steps in the development process to extract the model and to integrate the legacy component in a system.

An integration is successful if the communication with the environment is faultless. Furthermore, it is important to proof that dependent on the control flow (the protocol behavior) the needed control behavior is executed. Hence, an approach for hybrid systems is required which takes safety requirements into account.

In [GHH08a] we present an approach which enables, in contrast to existing approaches (e. g. [RMSM09, BJR06]), an integration of legacy components which takes safety requirements into account. The approach is limited to legacy components which support to monitor the current state information at the interface of the component. As we have presented in [BGH^{+}08] this approach works well for advanced applications in the RailCab project[1] and applications which propose the AUTOSAR[2] standard.

We extend this approach by taken legacy components into account which support only in- and output information in form of messages at the interface (typically known as black-box components) and legacy components which support the message information at the interface as well as the source code (known as white-box components). Especially the black-box case is very often given by applications in the automotive domain (e. g. [BB08]). As in our previous approach

[1]http://www-nbp.uni-paderborn.de/
[2]http://www.autosar.org/

[GHH08a], we learn and check (iteratively) the control flow of the communication with respect to the context of the legacy component (the modeled counterpart of the legacy component). Furthermore, we integrate the possibility to identify controller behavior by classical linear system identification approaches [Ise92]. We identify the controller behavior in form of transfer functions. If all transfer functions are known for each state of the control flow of the communication, we can identify reconfigurations of the controller behavior. Hereby, we support an utility which enables an early conflict recognition also for the engineers. Actual the integration of legacy controller behavior is done at the very end of the development process, in the system integration phase [BN03].

In the next section, we present a sketch of our approach. We present the relevant parts of MECHATRONIC UML [GHH$^+$08b], the model-driven engineering approach in which we integrate legacy components, and the required extension for integrating legacy components. After reviewing the relevant related work in Section 3, we present our integration approach in Section 4. The presented techniques are developed in the project group ReCab [MB09]. We conclude with a summary and future work.

2. SKETCH OF APPROACH

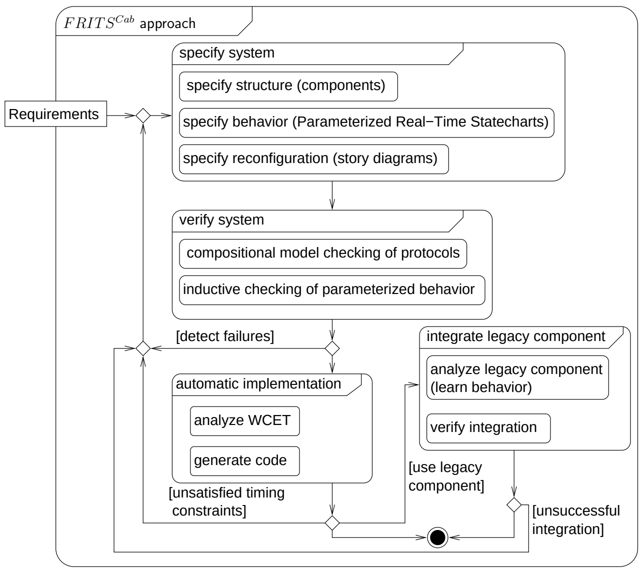
The approach is implemented in the Fujaba Real-Time Tool Suite. Figure 1 gives an overview of the development process of $FRiTS^{Cab}$[3]. We first start with the specification of the system (specify system). As in the MECHATRONIC UML approach [GHH$^+$08b], the system is specified in a compositional manner. We start with specifying the structure in form of components and the behavior by real-time coordination pattern (with parameterized real-time statecharts). The reconfiguration behavior of (software, controller) components and its ports is described by story diagrams. Based on this specification we use model checking to proof the correctness of the specified system with respect to (safety and liveness) constraints (verify system). If the system is correct, we generate automatically c++ code for simulation purposes as well as for the real physical system (automatic implementation).

As there are a lot of examples in the context of networked mechatronic systems, which require a dynamic structure and an adaptation to the changing environment, the MECHATRONIC UML approach focuses especially on the support of self-adaptation. An often used example is a convoy of autonomous vehicles for reducing the energy or for extending the comfort of a driver (of a car) or for driving cars through a road work in a safe manner[4]. As oftentimes the collaboration between a flexible number of participants is required in these examples, we extend the MECHATRONIC UML approach in [HHG08] such that we can model collaborations between components which include structural adaptation in form of new or removed ports as well as multi-ports with an open number of participants. To enable this kind of specification we extend the classical state based specification of behavior by Story Diagrams. $FRiTS^{Cab}$ also enables the code generation and worst case execution time analysis (WCET) of dynamic collaborations in an appropriate manner. The basic idea is that we can adapt the upper and lower limit of the required resources during runtime. The approach guarantees predictability, which is of paramount importance for real-time systems. As the focus of this paper is especially on the reverse engineering part, we refer to [MB09] for more details.

[3]*FRiTS*: *F*ujaba *R*e-Eng*i*neering *T*ool *S*uite, *Cab*: short form of RailCab, our (real) application which represents a mechatronic system.

[4]Our test bed for proving such examples is the RailCab project (http://www-nbp.uni-paderborn.de/).

After the automatic implementation step is finished, we can integrate legacy components (integrate legacy components). A side effect of the integration is an appropriate representation of legacy components in the MECHATRONIC UML approach. Additionally to the components whose behavior is specified by real-time statecharts, $FRiTS^{Cab}$ extends the structure by legacy components (black-box components) as shown in Figure 3. After we have specified, verified and synthesized the system, we can than proof the correctness of the integrated legacy components. The integration in the MECHATRONIC UML approach also enables to enhance the evolution of legacy components by using our modeling approach for self-adaptive systems. E. g. we can wrap the legacy component and add self-adaptive behavior in form of parameterized real-time statecharts and its reconfiguration to enable the communication to an arbitrary number of communication partners. The wrapping on the model- and code-level is still a manual task. If the integration is successful, the development is complete. The presented process is idealized for demonstration purposes. It is also possible to start the integrate legacy components phase on a partially specified system.



FRITS^Cab approach
Requirements
specify system
specify structure (components)
specify behavior (Parameterized Real−Time Statecharts)
specify reconfiguration (story diagrams)
verify system
compositional model checking of protocols
inductive checking of parameterized behavior
[detect failures]
integrate legacy component
analyze legacy component (learn behavior)
verify integration
automatic implementation
analyze WCET
generate code
[use legacy component]
[unsatisfied timing constraints]
[unsuccessful integration]

Figure 1: Overview of $FRiTS^{Cab}$

3. STATE OF THE ART

Abstraction - White-box Approach. Abstraction is an important technique for handling the state explosion problem of model checking. Counterexamples are often used to refine an abstract model. The upper approximation is refined, if some behavior in the approximation which is not present in the original model is the cause of a counterexample. When this happens, it is necessary to refine the abstraction so that the behavior which caused the erroneous counterexample is eliminated. Based on white box knowledge like the program variables, the approach is to find a model of the system with a good abstraction to reduce verification efforts. First, it is started with an over-approximation of states (states are reduced to one). Then, the model is refined as long as erroneous counterexamples are eliminated. A number of approaches are investigating this problem, like [CKL04, BHT06].

These approaches are based on white box information. Hence, no tests are required and these approaches do not require to consider the possible alphabet of the system, which is the basis for a black box approach. An interaction to the environment of the system, e.g. in the form of a context, is not considered, as well as time. Our white-box approach especially focusses on the correct mapping of the context model to code to enable a compositional checking of a real-time system on code level by using an existing source code model checker like CBMC [CKL04].

Regular Inference - Black-box Integration. In regular inference systems are viewed as black boxes. It is assumed that the considered black box system can be modeled by a deterministic finite automaton (DFA). The problem is than, to identify the regular language $\mathcal{L}(M)$ of the system $\mathcal{M}$. Learning algorithms are used to identify the regular language. A *Learner*, who initially only knows the alphabet Σ^* about $\mathcal{M}$, is trying to learn $\mathcal{L}(M)$ by asking queries to a *Teacher* and an *Oracle*. $\mathcal{L}(M)$ is learned by *membership queries* which asks the *Teacher* whether a string $w \in \Sigma^*$ is in $\mathcal{L}(M)$. Further, an *equivalence query* is required to ask the *Oracle* whether the hypothesized (learned) DFA $\mathcal{A}$ is correct ($\mathcal{L}(A) = \mathcal{L}(M)$). The *Oracle* answers yes if $\mathcal{A}$ is correct, or else supply a counterexample. Typically, the *Learner* asks a sequence of membership queries and builds a hypothesized automaton using the observed answers. When the *Learner* determines that the hypothesized behavior is stable an equivalence query is used to find out whether the behavior is correct. If the query is successful the *Learner* has succeeded, otherwise the returned counterexample is used to revise $\mathcal{A}$ and perform further membership queries until deriving the next hypothesized automaton, and so forth. A common assumption for equivalence checking is that A has at most as many states as M [BGJ+05]. If this is the case, the learned behavior is a correct representation of the legacy component (otherwise, it is not know).

Angluin's Algorithm. The most widely recognized regular inference algorithm is L^* developed by Angluin [Ang87]. The algorithm organizes the information obtained from queries and answers in a so called observation table. The observation table regards each string as consisting of a prefix and a suffix. The prefixes are indices of rows and the suffixes indices of columns in the table. A prefix is a string which leads to a state in the system, and a suffix is used to distinguish prefixes that lead to different states. A number of approaches exist, which are based on Angluin's [Ang87] learning algorithm. Some approaches, like [BJLS03, HNS03, BJR06], extend the algorithm of Angluin to get better runtime behavior in specific applications or domains. Other approaches use Angluin's algorithm and add technologies like testing or verification. Despite [GPY02], most approaches rather try to synthesize the whole behavior and than finding conflicting situations. However, our approach considers especially the collaboration (context) between the environment and the legacy component. Thus, the whole behavior of the legacy system is not required but only the relevant part for the collaboration. Similar to [GPY02] our black box approach is able to find real errors after each learning step. The presented approaches, besides [GJP06], did not consider time (constraints).

System identification. The (discrete) protocol behavior and the (continuous) controller behavior yields a hybrid system. Hybrid system identification approaches are well known in the control theory community, but does not scale very well [Lju08]. Hence our approach addresses this problem by reducing the hybrid system identification problem to a standard linear system identification problem as we split the identification in two steps. The first step is to identify the protocol behavior based on the approaches presented previous in this section and the second step is to identify the control behavior of the identified states of the protocol behavior.

4. INTEGRATION OF LEGACY COMPONENTS

Reverse engineering [CI90] is the discipline which takes the analysis and understanding of (legacy) software systems into account. As reverse engineering is a time intensive task, (semi-) automatic approaches are required for analyzing the relevant information. In our case, an approach is required which can learn the relevant behavior information for integrating a legacy component by proving that the specified constraints for the integration are fulfilled. We can distinguish between white-box legacy components (the source code and all relevant information for the execution of the legacy component are known) and black-box legacy components (only the relevant information of the interface and the execution of the legacy component are known). In the following, we first describe an approach for taking white-box legacy components into account. Afterwards, we consider black-box legacy components and the integration of system identification to enable the reverse engineering of control behavior. We will close this section with a short discussion.

Figure 2 gives an overview of the integration approach. The integration for the white-box- and black-box- approach can either show a refinement of a abstract role, if specified, or can show a correct communication with the context. Then, based on the learned state behavior the controller behavior is learned by system identification. Hence, it is possible to identify a reconfiguration of controller behavior. Figure 3 shows a component architecture including a legacy component, which is analyzed including the context behavior of Figure 4 in the following.

White-Box. A precondition for this approach is, that the source code (in our case C) of the legacy component is avail-

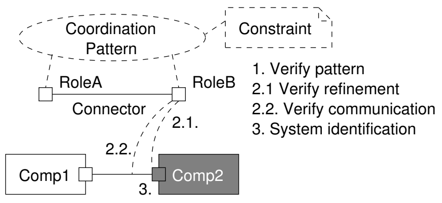


Figure 2: Overview of integration

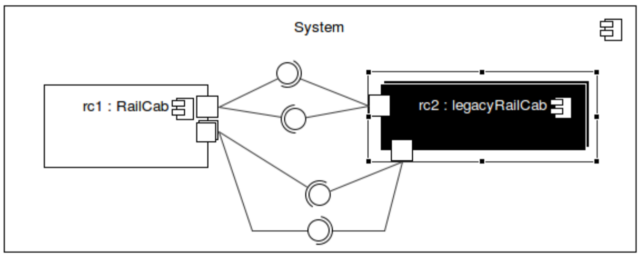


Figure 3: (Legacy-) Component Architecture

able. Furthermore, we have to know the interface procedures which are responsible for the communication. The basic idea is to use a source code model checker like BLAST[5] or CBMC[6] to enable a compositional formal verification or refinement on the source code level. The approach supports a C code legacy component and a C code component of the context or the C code of the abstract modeled role (of the legacy component) as input.

In a first step, we have to map the (abstract) model to source code to enable the required check (with the legacy component). The mapping has to preserve the execution semantics of the model. Hence, one possible (deterministic) path of the model has to be mapped to the source code. As the considered kind of systems are reactive once the generated system is executed periodically (that is also the case for the legacy component). The WCET (see Section 2) of a transition has to satisfy the specified deadlines. Within a period a task can be executed undeterministically by the scheduler. Furthermore, the periods of the legacy system and the model can be of different length.

Our Tool Suite supports a C code generation which considers the discussed requirements. Based on the generated code and a runtime framework which encapsulates the communication calls as well as a simulated time we can start the proof with a source code model checker. Conceptually the BLAST model checker fits best as this model checker supports a good abstraction based on lazy abstraction [BHT06]. But, in our evaluation BLAST had a lot of problems with bounded arrays which are required for the communication. The bounded model checker CBMC [CKL04] supports the most C constructs and also supports the required bounded arrays. Hence, we use CBMC for our evaluation. Figure 4 shows a part of the behavior which we have verified on the code level. The used model checker does only scale up for small examples. The cause is not the number of states but rather the number of non-deterministic variables due to a correct mapping of the scheduling and timing. E. g. the execution of a task can be arbitrary within a period. To have a realistic mapping, we use the concept of a non-deterministic variable for the starting point of the execution of a task. Non-deterministic variables are supported by BLAST as well as CBMC.

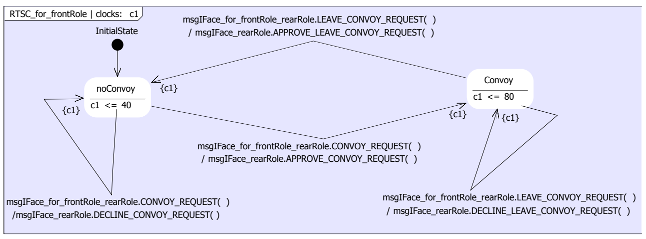

RTSC_for_frontRole | clocks: c1
InitialState
msgIFace_for_frontRole_rearRole.LEAVE_CONVOY_REQUEST( )
/ msgIFace_rearRole.APPROVE_LEAVE_CONVOY_REQUEST( )
noConvoy
c1 <= 40
{c1}
{c1}
Convoy
c1 <= 80
{c1}
{c1}
msgIFace_for_frontRole_rearRole.CONVOY_REQUEST( )
/ msgIFace_rearRole.APPROVE_CONVOY_REQUEST( )
msgIFace_for_frontRole_rearRole.CONVOY_REQUEST( )
/msgIFace_rearRole.DECLINE_CONVOY_REQUEST( )
msgIFace_for_frontRole_rearRole.LEAVE_CONVOY_REQUEST( )
/ msgIFace_rearRole.DECLINE_LEAVE_CONVOY_REQUEST( )

Figure 4: Verified behavior

Black-Box. Preconditions for the black-box integration are the interface procedures which are responsible for the communication and all other relevant information for the execution of the legacy component (e. g. like the period). We extend the existing work of [Ang87] by considering the context in form of a possible direct input for the membership queries (see Section 3) by verifying the learned behavior with the modeled behavior after each learning step. If the counter example is also true on the code level the integration is unsuccessful. Otherwise, if the learned behavior has been determined[7] and the model checking is successful, the integration is correct if the number of states are an upper bound of the state number of the legacy component [Ang87]. Because of the known counter part of the legacy component, the context or the abstract behavior, we can determine the number of states by the number of states of the modeled behavior. In most cases this should be a good approximation as in the considered domain with hard real-time requirements, typically every protocol consists of a watchdog pattern. That means, after each call an acknowledgment within a specified time is required which is an indication of a state. Furthermore, we consider timing (constraints) taken the known period of the legacy component into account.

The implemented concepts (including a caching for the membership queries) yield fast evaluation results. We have evaluated our approach by the presented behavior in Figure 4 and test models up to 15 states. The time for learning the correct behavior for 15 states is about two minutes. In [May08], we have identified coordination patterns in mechatronic systems. The number of states are about 10 for a pattern (role). Hence, the application of this approach to mechatronic systems is given.

System Identification. Before identifying the controller behavior, the state behavior has to be known. Hence, we require that the learning of the state behavior described above is successful. This approach is not applicable with the white-box approach as the applied tools do not support an exter-

[5]http://mtc.epfl.ch/software-tools/blast/
[6]http://www.cprover.org/cbmc/

[7]That is the case if the equivalence query is successful (see Section 3)

nally visible model of the checked systems (accordingly, an extension of the CBMC or Blast tool is future work)[8].

As described in the introduction, besides the communication the different (controller) modes are of importance for a mechatronic system, as well. System identification [FPW98], is the approach which enables the identification of a controller algorithm. This is done by simulation. We can simulate each path of the learned behavior and for each state we can identify the controller behavior. The input of the system identification is a specified test trajectory or a realistic run in its environment. Based on the input and output behavior the transfer function of the controller is identified for linear systems. If the transfer functions are known, we can identify reconfigurations by different transfer functions. This approach supports the engineer in integrating mechatronic components into early development phases. Typically the engineers test the legacy components (controllers) only in hardware-in-the-loop scenarios or the real environment later in the development process. In order to get realistic simulation runs we can use our deterministic replay framework [GH06]. In $FRiTS^{Cab}$ we have integrated the MATLAB System Identification Toolbox[9].

Discussion. The $FRiTS^{Cab}$ approach extends our former work and has the benefit that it could pin-point real failures in the test step which are no false negatives right from the beginning. In addition, it can also prove the correctness of the integration for an abstract behavioral model of the legacy component without learning the whole legacy component by only checking possible integration problems for the explicit given context or abstract behavior. This is true for the white-box case. For the black-box approach we require an upper number of the states of the relevant behavior for the integration of the black-box component. As explained in the previous paragraph, a good approximation is the number of states of the context or abstract behavior. The presented integrated system identification approach enables also to identify the controller behavior. Together with the learned state behavior, we can identify a restricted class of hybrid systems which change the configuration (the controller behavior) in an event triggered manner. For the complete class of hybrid systems system identification is not possible as a reconfiguration can be triggered at any time by a continuous value [Lju08]. In our former work, we have shown that the considered class of hybrid systems are of relevance for mechatronic systems [GBSO04]. In this paper, we present only a one port to one port integration. If the legacy component has a dependency to more than one port the parallel product of the dependent ports is used.

5. CONCLUSION AND FUTURE WORK

In this paper, we presented an overview of $FRiTS^{Cab}$, a tool for the re-engineering of mechatronic systems. The tool is integrated into the Fujaba Real-Time Tool Suite. $FRiTS^{Cab}$ especially enables the integration of white- and black-box components in a MECHATRONIC UML architecture by taken safety requirements into account. Currently, we especially focus on the evaluation of the integration approach by industrial applications.

[8]But, constraints of the controller behavior can be checked by the presented tools based on the identified states of the protocol behavior.

[9]http://www.mathworks.com/products/sysid/

6. REFERENCES

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[Bal00] Helmut Balzert. *Lehrbuch der Softwaretechnik - Software-Entwicklung*. Spektrum-Akademischer, 2 edition, 2000.

[BB08] Gerd Baumann and Michael Brost. Testverfahren für elektronik und embedded software in der automobilentwicklung Ü eine Übersicht. In Bernhard Schätz, Holger Giese, Ulrich Nickel, and Michaela Huhn, editors, *Proc. of the Dagstuhl-Workshop: Model-Based Development of Embedded Systems (MBEES), 7.3.-12.3.2008, Schloss Dagstuhl, Germany*, number 2008-02 in Informatik-Bericht, pages 13–20, Technische Universität Braunschweig, April 2008.

[BGH^{+}08] Christian Brenner, Holger Giese, Stefan Henkler, Martin Hirsch, and Claudia Priesterjahn. Integration of legacy components in mechatronic uml architectures. In Uwe Aßmann, Jendrik Johannes, and Albert Zündorf, editors, *Proc. of the 6^{th} International Fujaba Days 2008, Dresden, Germany*, pages 52–55, September 2008.

[BGJ^{+}05] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the correspondence between conformance testing and regular inference. In *Fundamental Approaches to Software Engineering*, volume Volume 3442/2005, pages 175–189. Springer Berlin / Heidelberg, 2005.

[BHT06] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy shape analysis. In *In CAV'2006: Computer Aided Verification, LNCS 4144*, volume 4144, pages 532–546, 2006.

[BJLS03] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to Angluin's learning. In *Proceedings of the International Workshop on Software Verification and Validation (SVV 2003)*, volume 118 of *Electronic Notes in Theoretical Computer Science*, pages 3–18, dec 2003.

[BJR06] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.

[BN03] Bart Broekman and Edwin Notenboom. *Testing Embedded Software*. Addison-Wesley, 2003.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[FPW98] G. F. Franklin, J. D. Powell, and M. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley Longman Publishing Co., Inc., 3 edition, 1998.

[GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver 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.

[GH06] Holger Giese and Stefan Henkler. Architecture-driven platform independent deterministic replay for distributed hard real-time systems. In *Proceedings of the 2nd International Workshop on The Role of Software Architecture for Testing and Analysis (ROSATEA2006)*, pages 28–38, New York, NY, USA, July 2006. ACM Press.

[GHH08a] Holger Giese, Stefan Henkler, and Martin Hirsch. Combining Compositional Formal Verification and Testing for Correct Legacy Component Integration in Mechatronic UML. In Rogério de Lemos, Felicita Di Giandomenico, Cristina Gacek, Henry Muccini, and Marlon Vieira, editors, *Architecting Dependable Systems V*, volume 5135 of *LNCS*, pages 248–273. SPRINGER, 2008.

[GHH^{+}08b] Holger Giese, Stefan Henkler, Martin Hirsch, Vladimir Roubin, and Matthias Tichy. Modeling techniques for software-intensive systems. In Dr. Pierre F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*, pages 21–58. Langston University, OK, 2008.

[GJP06] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of event-recording automata using timed decision trees. In *CONCUR 2006 Concurrency Theory*, volume Volume 4137/2006, pages 435–449. Springer Berlin / Heidelberg, 2006.

[GPY02] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume Volume 2280/2002, pages 269–301. Springer Berlin / Heidelberg, 2002.

[HHG08] Martin Hirsch, Stefan Henkler, and Holger Giese. Modeling collaborations with dynamic structural adaptation in mechatronic uml. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 33–40, New York, NY, USA, 2008. ACM.

[HNS03] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *Computer Aided Verification*, volume Volume 2725/2003, pages 315–327. Springer Berlin / Heidelberg, 2003.

[Ise92] Rolf Isermann. *Identifikation dynamischer Systeme*. Springer-Verlag, 1992.

[Lju08] Lennart Ljung. Perspectives on system identification. In *Proc. 17th IFAC World Congress*, Seoul, Korea, July 2008. (Plenary session).

[May08] Karl Alexander May. Identifikation von koordinationsmustern in autonomen mechatronischen echtzeitsystemen. bachelor thesis, University of Paderborn, September 2008.

[MB09] Christian Brenner Kathrin Bröker Uwe Pohlmann Manel Richtermeier Julian Suck Oleg Travkin Moritz Breit, Markus Böger. *Abschlussarbeit der Projektgruppe ReCab: Re-Engineering mechatronischer Systeme*, 2009. to appear.

[RMSM09] Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):307–324, October 2009.

Specification and Refinement Checking of Dynamic Systems[*]

Christian Heinzemann, Stefan Henkler
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[chris227|shenkler]@uni-paderborn.de

Albert Zündorf
Software Engineering Research Group
University of Kassel
Wilhelmshöher Allee 73
D-34121 Kassel, Germany
zuendorf@uni-kassel.de

ABSTRACT

Software is increasingly used in systems which have to support self* properties like self-adaptation, -management or -optimization. The key enabler for a consistent model-based development approach is refinement. Refinement facilitates to preserve properties of abstract models in more concrete models. Note, that abstract models are of paramount importance to formal verification in complex safety critical systems. Despite the increased significance of self* properties in the last years, surprisingly there is a lack in support of refinement techniques being integrated in a model-based development approach. We present a modeling approach, called Timed Story Charts, which supports a flexible specification of properties like self-adaptation, and furthermore, we will present an integrated refinement check.

1. INTRODUCTION

Advanced software systems, like mechatronic systems, increasingly exhibit self* properties like self-adaptation, -management or -optimization. That implies software reconfiguration at runtime which increases the complexity of the software additionally. As these systems are often used in a safety critical environment, formal verification on abstract models is required to ensure a proper functioning of the software. Despite the increased significance of self* properties in the last years, surprisingly there is a lack in support of refinement techniques being integrated in a model-based development approach. There are some approaches for modeling the structural aspects of reconfiguration or the behavioral aspects but none of them take into account both aspects [BCDW04].

Based on our approach Mechatronic UML [GHH$^+$08], we present a modeling approach, called Timed Story Driven Modeling, an extension of Story Driven Modeling [Zün01], supporting a flexible specification of reconfiguration. Further on, we will present an integrated refinement check.

Mechatronic UML is based on a methodical decomposition of the embedded software and its constituent components. This supports compositional verification. Because of the dynamics in the behavior as well as the dynamics in the number of participating components, there is a need for modeling support for dynamic structures. This is also an existing problem of UML-components and -parts as e. g. creation and deletion of a part and its (delegated) port is not supported in UML. Similarly, UML does not address the question whether an embedded component (part) is a correct refinement of the protocol behavior of the surrounding component.

The presented Timed Story Driven Modeling approach supports the specification of complex dependencies of evolving behavior and it supports the specification of timing constraints for hard real-time systems like timed automata [AD94]. So, it combines the power of story driven modeling as well as timed automata, the quasi standard for the specification of real-time behavior. Hence, a consistent formalism is defined which is required by the analysis of timed behavior with complex dynamic changes at runtime. In contrast to former work [HHG08], we have a well defined consistent formalism and did not have to reason about different formalisms (which is difficult and could be error prone). Therefore, we only have to reason about analysis techniques for one formalism and not for different ones (e. g. statecharts and graph transformation systems). In this paper, we present only an overview of the Timed Story formalism and refinement checking. For more details, we refer to [Hei09].

In the following section, we present our Timed Story Driven Modeling approach. In Section 3, we present a refinement check for the Timed Story Driven approach. Related work is discussed in Section 4. We conclude with a summary and future work in Section 5.

2. TIMED STORY DRIVEN MODELING

To specify a system being able to dynamically change the structure (and therefore also the behavior) at runtime, we specify the structure of the system with UML 2.0-components and a many to many association between components (using so-called multi ports) and / or an to many association to its embedded components (called multi parts, see Section 2.1). The reconfiguration of the architectural parts like components, parts or ports are specified by (Timed) Story Diagrams and (Timed) Story Pattern (see Section 2.2 and Section 2.3). It seems, that in some cases we can specify the timing constraints of a reconfiguration by the behavior which triggers the reconfiguration (in form of timing constraints for the trigger). Hence, the timed ver-

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

sion of Story Pattern and Story Diagrams are in some cases optional. The behavior of the system, which also triggers the reconfiguration, is specified by Timed Story Charts (see Section 2.4).

2.1 System architecture

The system architecture is specified by UML 2.0-components and -parts. For each component diagram, a class diagram is automatically synthesized. The class diagram includes classes for each component and its ports, for each embedded part, and for all delegations and assemblies. The structure of the class diagram is based on the meta model of the component diagram (see Figure 1). An example class diagram is shown in Figure 2.

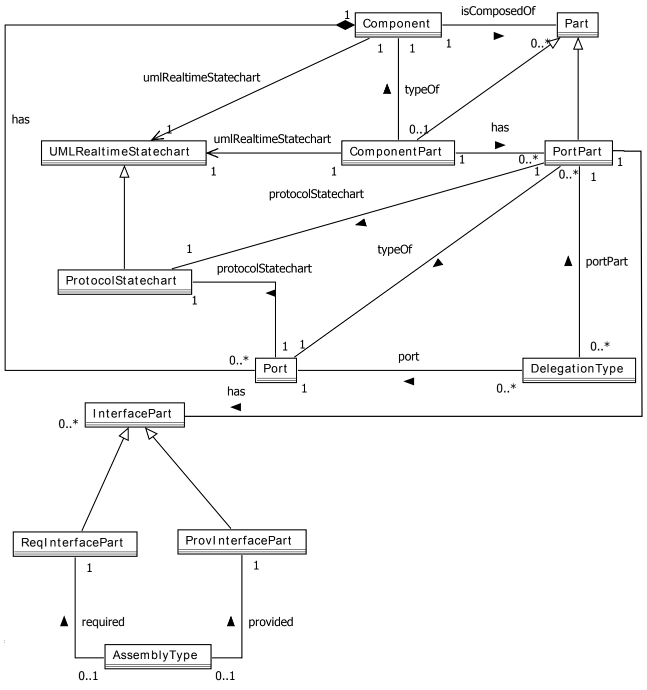


Figure 1: Component and parts meta model

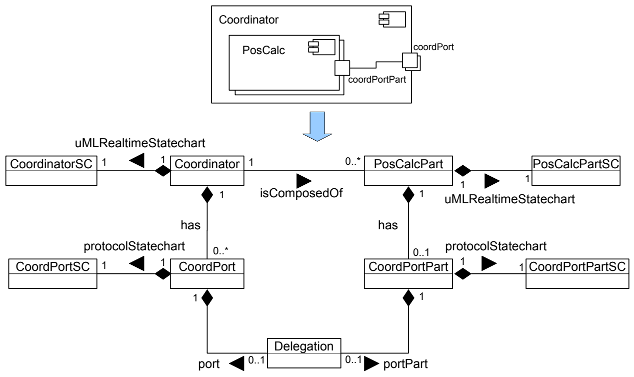


Figure 2: Example class diagram

2.2 Timed Story Pattern

In this section, we extend Story Pattern by time. Similar to the approach described in [Hir08], which extends the groove syntax and semantics by time, the timing concept of Timed Story Pattern is based on the semantics of Timed Automata [AD94]. We therefore support the specification of clocks, clock resets, time guards, and invariants.

Clocks are described by *clock instances* (see Figure 3). That means, clocks are represented by objects. Clocks are defined by a clock instance and its links to the objects of a graph. A definition of a clock instance for an edge is indirectly specified by the related objects of the edge.

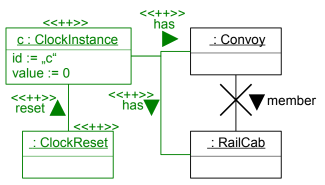


Figure 3: Defining a clock c

Clock resets are also modeled by objects which have a link reset to a clock instance. A clock reset object is instantiated at the same time as the associated clock instance. A *time guard* is implemented by a "standard" Story Pattern condition by referring to clock values. A precondition for specifying a time guard is, that the referred clock instance has to be bound in the Story Pattern. *Invariants* are specified by a special Story Pattern having no right hand side and the referred clock instance has to be bounded.

2.3 Timed Story Diagrams

The difference between a Timed Story Diagram and a Story Diagram is that a pattern of a Timed Story Diagram is a Timed Story Pattern. The semantics is the same as for Story Diagrams.

2.4 Timed Story Charts

The Timed Story Chart formalism supports abstract states, time constraints, and integrates dynamic adaptation by triggering a reconfiguration specified by (Timed) Story Pattern or (Timed) Story Diagrams.

Figure 4 shows the meta model of Timed Story Charts to support the specification of states and timing by objects. Transitions are implicitly implemented by rules[1]. In contrast to [Zün01], we did not use a framework for the execution semantics of the Story Chart as a reachability analysis would be difficult due to the single method executing the transitions. Hence, a transition would not be (easily) identifiable. Instead, we specify a story diagram which describes the execution semantics. In more detail, we explain in the following an overview of the syntax of some elements of Timed Story Charts and its execution semantics. We focus on the implementation of some specific statechart constructs. Clocks, guards, time invariants, time guards, clocks resets are implemented in Timed Story Pattern (see Section 2.2). Deadlines are implemented by the use of invariants and time guards and therefore are not discussed anymore.

States are represented by an extra object of type State. The name of the state is implemented by an attribute. *AND*

[1]An explicit transition object would lead to extra computations by the analysis

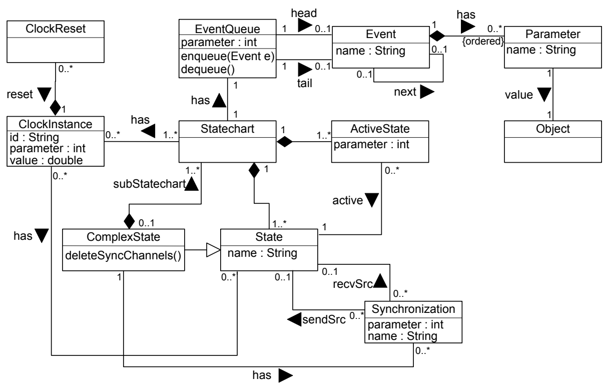


Figure 4: Meta model for the mapping of real-time statecharts to story diagrams

States are implemented by the class ComplexState. ComplexState can embed a set of statecharts. The *Active State* is implemented by an ActiveState object having an association to the active state. To differ between different instances of a Statechart (e.g. instances of a parameterized Statechart), we add an attribute parameter to the ActiveState object. If a statechart is instantiated more than once, an ActiveState object for each statechart is instantiated with a specific parameter. With this technique it is possible to manage the statechart instances of multi-parts and multi-port (see Figure 5).

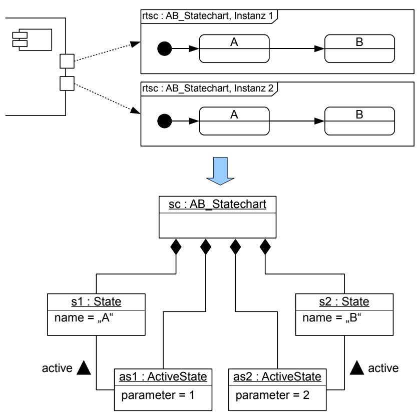


Figure 5: Example of a mapping of a statechart to a graph

An event is implemented by the Event type. The name of the event is implemented by an attribute of Event. Parameterized events are implemented by an ordered set of parameters by the type Parameter. The value of the parameter is specified by an association to a corresponding object. An event-queue is associated to each statechart (instance). An example of a trigger event a is shown in Figure 6. A raised event is implemented by an event object with modifier $<< ++ >>$. Raised events have to be added to the event queue of the receiving statechart (instance) which is not shown in the figure. Further more, Figure 6 shows the implementation of transition by a Story Diagram. The syntax of a story is that of a Timed Story Pattern (see Section 2.2).

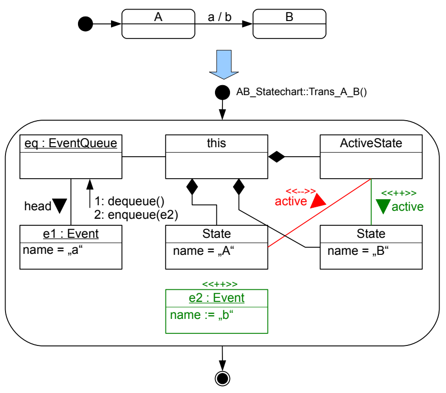


Figure 6: Events

Synchronization is implemented by an extra Synchonisation object. By a name attribute the name of the synchronization is implemented and the parameter attribute refers to a specific statechart instance. As the transitions which have to be synchronized have to fire simultaneous a joint story diagram is specified including both transitions. An example is shown in Figure 7.

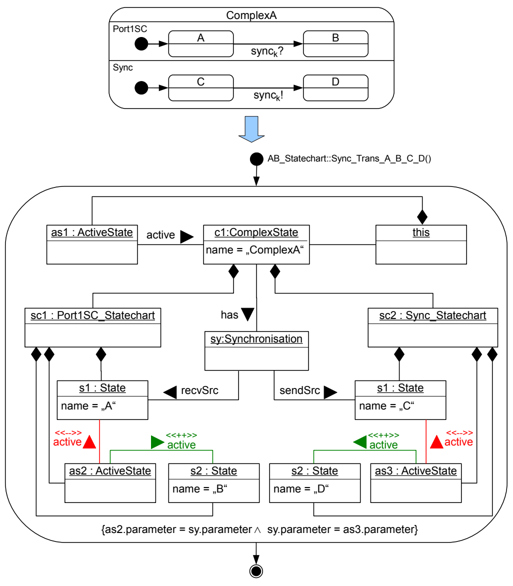


Figure 7: Synchronization

The syntactical mappings of (Real-Time) Statechart constructs defined above are now combined to a sequence of transformations (see Figure 8) defining the execution semantics of Timed Story Charts. Note, we discuss only a view relevant constructs in this paper. A complete definition of Timed Story Charts is presented in [Hei09]. A transition of a Real-Time Statechart is mapped to a modular Timed Story Chart with a set of stories. Hence, the exchange of the semantics is easy. As an example, we take into account the semantics of a transition without deadline. Figure 8 shows a schematic Timed Story Chart. The activities are stories. The first story analyzes whether the precondition is fulfilled. This includes binding the source state, the event trigger, and synchronization channels. Furthermore, all (time) guards are considered. If all bindings and (time) guards are fulfilled, the transition can fire (1. story). The 2. story takes the relevant events from the queue. The 3. story eliminates all synchronization objects from the source state. In story 4. the exitAction is executed. The side effect is executed in story 5. and the trigger events are eliminated. Next, in story 7. , raised events are created and clocks are reseted. Finally, in story 8. the synchronization channels of the target state are instantiated and, in story 9., the target state is entered.

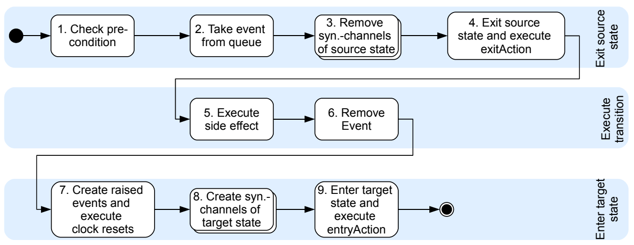

1. Check pre-condition
2. Take event from queue
3. Remove syn.-channels of source state
4. Exit source state and execute exitAction
Exit source state
5. Execute side effect
6. Remove Event
Execute transition
7. Create raised events and execute clock resets
8. Create syn.-channels of target state
9. Enter target state and execute entryAction
Enter target state

Figure 8: Execution semantics

3. REFINEMENT

Basically, we have two requirements for the refinement: 1) the external visible real-time behavior has to be fulfilled by the refined behavior and 2) the (formal) compositional verification results of the abstract behavior have to be preserved by the refinement (cf. Figure 9).

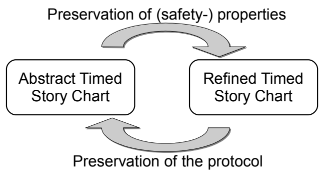

Preservation of (safety-) properties
Abstract Timed Story Chart
Refined Timed Story Chart
Preservation of the protocol

Figure 9: Requirements to the refinement

For supporting requirement 1), we require that each external visible trace consisting of events (in- and out-going) and its timing, is implemented by the refined behavior. That means each event supported by the abstract behavior has to be supported by the refined behavior[2] as well and the refined behavior has to react in the same time interval as the abstract one. In contrast to other definitions, we allow the refined behavior to have a more relaxed receive interval. For requirement 2), we require that each trace of the refinement is related to (simulated by) a trace of the abstract behavior. That means the abstract behavior simulates the refined and therefore compositional verification results are preserved [CGP00]. In [Hei09] a detailed definition is presented.

In the following we present the refinement check by first computing a reachability graph of the abstract and the refined Timed Story Chart (see Section 3.1). In principle the reachable graph could be infinite. As we take into account hard real-time systems, the reachable graph is finite as dependent behavior (statecharts) has an upper hard timing constraints or the behavior are independent and therefore only one possible instance of the behavior has to be considered. In [Hei09] we discuss an alternative approach which could also take into account an infinite system. Based on the reachable graphs, we illustrate an algorithm for checking the refinement in Section 3.2.

3.1 Reachability Analysis

The timed reachability analysis is based on the computation of reachability graphs as introduced in [Zün09]. Basically, the timed story patterns used to execute the timed story charts are transformed such that the matching and the rewrite step are separated into two operations. Then, the matching operation is embedded into a for each construct, searching for all possible matches of a given timed story pattern in the current graph. For each match, we use a library operation introduced in [Zün09] in order to create a copy of the current graph and then, the rewrite operation is applied to that graph copy. We do this for all story patterns that are enabled for the current graph. Thus, for a given start graph the expansion step described above computes the set of all possible successor graphs reachable with the available story patterns. We apply this expansion step to all reachable graphs as long as possible. During the expansion, we use a library provide isomorphism check [Zün09], to compare each new graph with all other derived graphs. Thereby, we identify and merge graphs that may be reached by different sequences of story pattern applications. During the application of timed story patterns, the timing constraints are maintained using a dedicated clock zone ([CGP00, p. 280]), cf. [Hei09]. Accordingly, handling of the clock zones is incorporated in the graph copy operation and especially in the graph isomorphism check. Thus, two timed graphs are considered isomorphic, if the graph structure is isomorphic and if the clock zones are equivalent.

Note, in general the computation of a reachability graph may not terminate. In our case, we model the execution of finite timed story charts and the timing constraints result in another restriction concerning the length of execution pathes.

3.2 Refinement Check

The refinement check is realized by a depth-first search (see algorithm 1). The algorithm checks for each event of the refined behavior if a corresponding event in the abstract behavior exist. First the algorithm starts with a (timed) reachability analysis as described in the last section. Then it is checked if for the starting states a structural refinement exist. If nodes exist which are not expanded a successor is checked (row 8) and expanded (row 9). For each successor it is checked if the successor is already known. If a successor is not known it is checked if an event is triggered or raised and, if so, it is checked if a corresponding trace exist. If a node is already known it could be the case that a circle is closed or two traces are joined. In both cases, if an event is triggered or raised it is checked if a corresponding trace exist. In cases of a circle, we check if the circle is well-formed. That means, we check if the circle has a corresponding trace.

4. RELATED WORK

In [Gie07] a refinement is defined for hybrid graph transformation systems which preserves verification results of the

[2]It is allowed that the refined behavior supports more events than the abstract behavior

abstract behavior. The focus is not, as in our case, to define a more relaxed refinement which enables a more flexible integration of possible refined behavior and it is not required that the external visible real-time behavior is still preserved by the refined behavior. [HT04] considers graph transformation systems for the specification of service oriented architectures. The presented refinement should preserve the external visible services. The approach did not take into account time and the ability to preserve verification results. [GRPS02] examine refinement for graph transformation systems based on an algebra but they did not take into account time.

Algorithm 1 Refinement check

```
1: function checkCorrectRefinement(TimedStoryChart abs, TimedStoryChart ref)
2:     absReach = startReachabilityAnalysis(abs)
3:     refReach = startReachabilityAnalysis(ref)
4:     success := checkStructureRefinement (absReach.initial, refReach.initial)
5:     OPEN.push(refReach.initial)
6:     while OPEN $\neq \emptyset \wedge$ success do
7:         n := OPEN.pop( )
8:         success := refReach.hasSuccessor(n)
9:         for all $n' \in$ refReach.expand(n) do
10:            if $n'$ is not known then                ▷ Case 1: new node
11:                OPEN.push($n'$)
12:                if $(n, n')$ has event $e$ then
13:                    success := checkPath($(n, n')$)
14:                end if
15:            else                                     ▷ Case 2: node is known
16:                if $(n, n')$ closed cycle then       ▷ a) Edge closes an circle
17:                    if $(n, n')$ has event $e$ then
18:                        success := checkPath($(n, n')$)
19:                    end if
20:                    success := isWellFormedCycle($n'$)
21:                else                                 ▷ b) Joining of two traces
22:                    if $(n, n')$ has event $e$ then  ▷ The same as case 1
23:                        success := checkPath($(n, n')$)
24:                    end if
25:                end if
26:            end if
27:        end for
28:    end while
29:    if success then
30:        success := checkCoverage(absReach)
31:    end if
32:    return success
33: end function
```

5. CONCLUSION AND FUTURE WORK

We presented in this paper the Timed Story Driven approach and a refinement check for Timed Story Charts, the behavioral specification language of the Timed Story Driven approach. We gave in more detail an overview of the syntax and semantics of Timed Story Charts. The presented refinement check, which preserves compositional verification results and the external visible real-time behavior, is based on a reachability analysis.

The expand of the reachability analysis is a manual task. Especially for timed systems, this could be error prone. Hence, a future task is to develop an automatic expand.

6. REFERENCES

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM.

[CGP00] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* January 2000.

[GHH$^+$08] Holger Giese, Stefan Henkler, Martin Hirsch, Vladimir Roubin, and Matthias Tichy. Modeling techniques for software-intensive systems. In Dr. Pierre F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*, pages 21–58. Langston University, OK, 2008.

[Gie07] Holger Giese. Modeling and verification of cooperative self-adaptive mechatronic systems. In *Reliable Systems on Unreliable Networked Platforms*, volume 4322 of *Lecture Notes in Computer Science*, pages 258–280. Springer Berlin / Heidelberg, 2007.

[GRPS02] Martin Große-Rhode, Francesco Parisi Presicce, and Marta Simeoni. Formal software specification with refinements and modules of typed graph transformation systems. *J. Comput. Syst. Sci.*, 64(2):171–218, 2002.

[Hei09] Christian Heinzemann. Verifikation von Protokollverfeinerungen. Master Thesis, Software Engineering Group, University of Paderborn, Nov 2009. to appear.

[HHG08] Martin Hirsch, Stefan Henkler, and Holger Giese. Modeling Collaborations with Dynamic Structural Adaptation in Mechatronic UML. In *Proc. of the ICSE 2008 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08),Leipzig, Germany*, pages 33–40. ACM Press, May 2008.

[Hir08] Martin Hirsch. *Modell-basierte Verifikation von vernetzten mechatronischen Systemen.* PhD thesis, University of Paderborn, Paderborn, Germany, September 2008.

[HT04] Reiko Heckel and Sebastian Thöne. Behavioral refinement of graph transformation-based models. In *Proc. of the ICGT 2004 Workshop on Software Evolution through Transformations (SETra 04)*, pages 139–151. Electronic Notes in Theoretical Computer Science, 2004.

[Zün01] Albert Zündorf. *Rigorous Object Oriented Software Development.* University of Paderborn, 2001.

[Zün09] Albert Zündorf. Model Checking the Leader Election Protocol with Fujaba. In *5th International Workshop on Graph-Based Tools (GraBaTs)*, July 2009.