TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

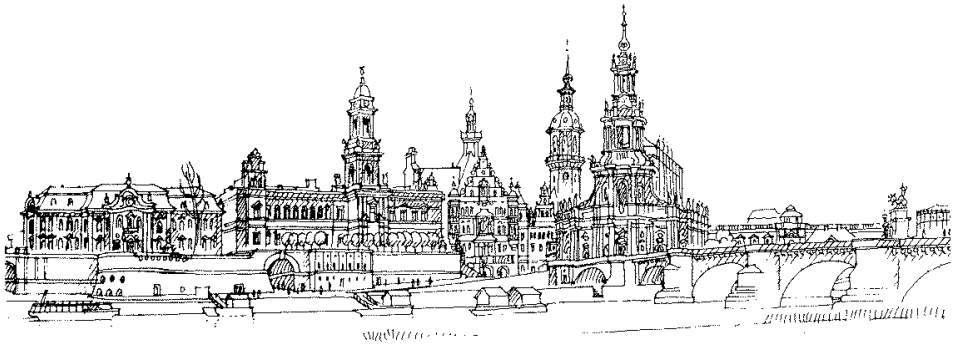
Technische Berichte
Technical Reports
ISSN 1430-211X

TUD-FI08-09 September 2008

Uwe Aßmann, Jendrik Johannes,
Albert Zündorf (Eds.)

Institut für Software- und Multimediatechnik

Proceedings of
6th International Fujaba Days



Technische Universität Dresden
Fakultät Informatik
D−01062 Dresden
Germany
URL: http://www.inf.tu−dresden.de/

Uwe Aßmann, Jendrik Johannes, Albert Zündorf (Eds.)

6th International Fujaba Days

Technische Universität Dresden, Germany
September 18-19, 2008

Proceedings

Volume Editors

Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Departement of Computer Science
Institute for Software and Multimedia Technologie
Software Technologie Group
Nöthnitzer Str. 46, 01187 Dresden, Germany
uwe.assmann@tu-dresden.de

DIpl. Medieninf. Jendrik Johannes
Technische Universität Dresden
Departement of Computer Science
Institute for Software and Multimedia Technologie
Software Technologie Group
Nöthnitzer Str. 46, 01187 Dresden, Germany
jendrik.johannes@tu-dresden.de

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

Program Committee

Program Committee Chairs

Uwe Aßmann (TU Dresden, Germany)
Albert Zündorf (University of Kassel, Germany)

Program Commitee Members

Uwe Aßmann (TU Dresden, Germany)
Jürgen Börstler (University of Umea, Sweden)
Gregor Engels (University of Paderborn, Germany)
Holger Giese (University of Paderborn, 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)
Andy Schürr (TU Darrmstadt, Germany)
Wilhelm Schäfer (University of Paderborn, Germany)
Gerd Wagner (University of Cottbus, Germany)
Bernhard Westfechtel (University of Bayreuth, Germany)
Albert Zündorf (University of Kassel, Germany)

Editor's preface

For more than ten years, Fujaba attracts a growing community of researchers in the field of software modelling. As its successful predecessors, the 6th International Fujaba Days offer a premises for Fujaba developers and users from all over the world to exchange experiences and drive Fujaba's future development.

The Software Technology Group of TU Dresden is active in the Fujaba community since 2005, sharing interests in areas such as UML and MOF modelling, OCL, and TGGs. Therefore, the group is proud to host the Fujaba Days 2008.

13 papers were received and accepted for the workshop after a careful reviewing process by the program committee. The papers are organized into four sections: Applications, Transforming and Debugging, Tool Integration, and Components.

The variety of topics shows that there is a growing interest in using Fujaba in new areas of software modelling and in conjunction with other modelling tools and environments. We are convinced that this years Fujaba Days will give the opportunity to drive these ideas forward and enable new fruitful collaborations among the community.

The PC chairs would like to thank the PC members for their careful work in reviewing the papers to ensure the quality of the Fujaba Days.

Uwe Aßmann and Jendrik Johannes
Organizers

Table of Contents

Applications

Transforming and Debugging

Tool Integration

Components

Simple Robotics with Fujaba

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

Fujaba is not only used for professional software development but also for teaching software development. We have used Fujaba successfully in our education efforts for a long time. In order to give students a better experience and feedback, we are preparing programming exercises with a simple robotic system. We are using the Lego Minstorms NXT robotics kit for that purpose. It consists of a micro-controller, motors, several sensors and of course, Lego parts to build different kinds of robots. The micro-controller can be either remote controlled by a PC or run deployed software. This paper describes how we use Fujaba to develop a software which controls a simple forklift robot, which solves the Towers of Hanoi game.

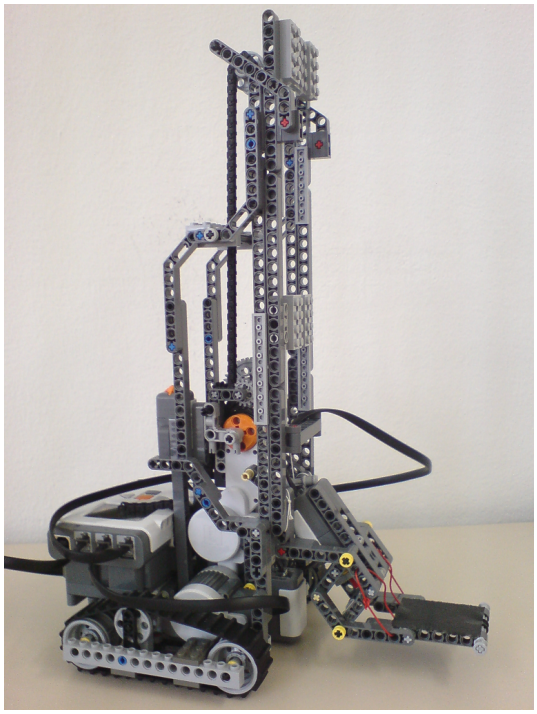


Figure 1: Forklift robot

1. INTRODUCTION

Fujaba is used for teaching software development successfully at Kassel University. In cooperation with the Gaußschule Braunschweig we developed a robot programming course with the predecessor product from Lego, the Mindstorms Robotics Invention System, which was released in 1998. Although it was possible to control a Lego forklift robot with Fujaba, the project faced lots of difficulties and obstacles. For example, the communication between the host PC and the mindstorms microcontroller was done by a infrared link, which was slow and unreliable. Furthermore, it was difficult to have more than one system in a room. More details can be found in [2]. Now we tried to use the newer Lego Mindstorms NXT robotics system [4], released in 2006. It provides improved hardware: a full-featured 32bit microprocessor, step counter motors, graphical LCD display and USB and bluetooth connectivity. The heart of the system, the so called NXT brick containing the micro-controller, can be either remote controlled by a PC or run deployed software. Four sensors and three motors can be connected to the brick.

This paper describes how we use Fujaba to develop a software which controls a simple forklift robot. Section 2 describes how we designed our forklift robot.

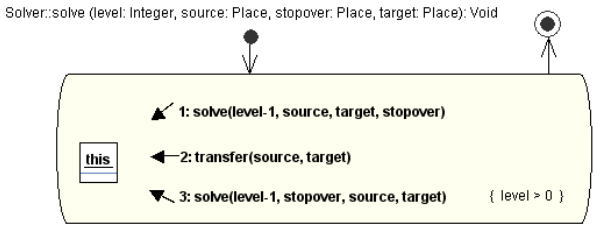


Figure 2: solve()-Method of Hanoi

At the practical side, our forklift robot should solve the Towers of Hanoi game. This is a well documented mathematical game or puzzle which consists of three rods, and a number

of disks of different sizes which can slide onto any rod. The puzzle starts with the disks stacked in order of size on one rod, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack step by step to another rod. Only one disk may be moved at a time. Each move consists of taking the top-most disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod. No disk may be placed on top of a smaller disk. In our approach, we replaced the disks by wooden blocks and don't use any rods. The blocks just get piled directly onto each other. The recursive software solution is very simple, see Figure 2. The *solve()*-method gets called recursively, *transfer()* simply moves the top disc from the source to the target stack. [1] discusses this recursive solution in detail. To carry that software solution into the real world, each time a disc is moved, the robot should drive to that certain block, grab it, drive to the destination and discard the block.

To keep the control software separately from the concrete problem, we decided to implement it as software library. Figure 3 shows the class diagram of our *FujabaNxtLib*, which is a Fujaba project. The central class is *FNXT* which represents the NXT Brick. The Motor class and the several kinds of Sensor classes are self explanatory. Each sensor class has a corresponding listener interface (not shown). The class *FNavigator* controls the two driving motors and provides methods for moving and turning the robot.

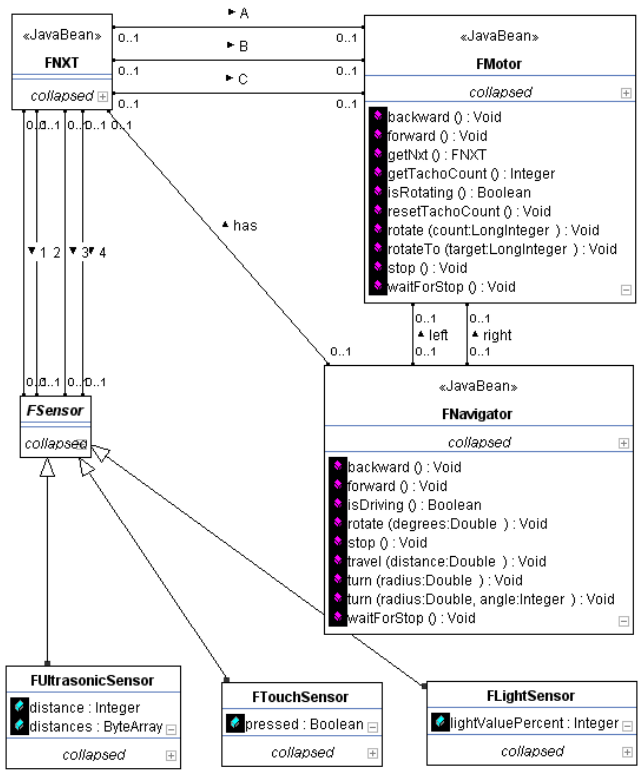


Figure 3: Class Diagram of the Fujaba NXT Library

2. THE ROBOT HARDWARE

It turned out that building the forklift itself was quite difficult. Our early Lego models did not turn properly, were unstable or had a bad center of gravity. Figure 1 shows our current forklift robot. We decided to use two chains instead of wheels so the robot rotates within its footprint. This way, the software may easily predict the turning, which is necessary for exact navigation. The third motor actuates the fork via a long chain and cord. The NXT lego motors contain a step counter: It is possible to turn the motor just a certain amount of rotation, with the accuracy of one degree. So, we don't need a sensor to detect the upper bound for lifting the fork, we use a fixed constant value and synchronize with the ground. Touching the ground resets the step counter. The robot uses only two sensors: a light sensor is mounted just above the ground and can detect changes in the ground contrast. This can be used mark certain places for the robot. The sensor is mounted in the front center of the robot. It can be used to follow the right edge of a black tape line on the ground: When detecting black, the robot turns lightly right while driving forward, otherwise left. The second sensor is mounted on the fork and detects when the fork touches the ground or when it touches an object while driving forward. The trigger mechanism at the fork passes vertical and horizontal force to a single touch sensor. Figure 4 shows the fork sensor mechanism in detail.

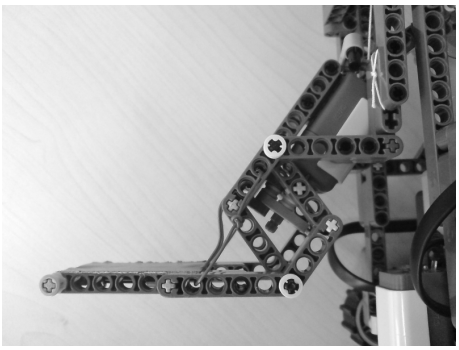


Figure 4: Fork sensor mechanics

3. SOFTWARE ARCHITECTURE

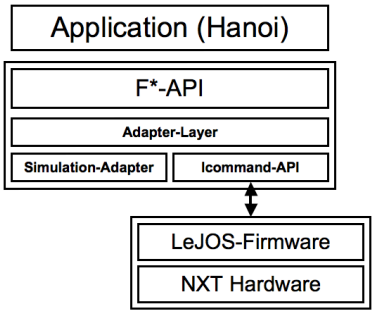


Figure 5: Software architecture layers of the Fujaba/NXT system

We are using the *LeJOS open source java firmware* (see [5]), which includes a Java virtual machine and a Java Micro-Edition-like API. The original Lego firmware has to be replaced by LeJOS. Additionally, LeJOS provides a remote control API called *ICommand* for PCs/Java2SE. We decided to use the latter, and control the NXT remotely over the bluetooth connection with software running on a PC host.

The sensor software components from Lego don't support any listener callbacks on changes directly, it is just possible to ask the sensors for their current value. So, we have to constantly poll all sensors and create events when the sensor values change. This is necessary because the controlling software, which runs on the host, needs to react quickly on events, for example by stopping a motor. Polling the sensors takes only 10-50ms each time, so the polling is performed periodically at a fixed interval. Fortunately, this does not interfere with any control commands. These is a radical improvement over the old Mindstorms system, which had a poll latency of 100-200ms per sensor.

We are planning to replace the poll mechanism by a event notifier running directly on the NXT's micro-controller, which call their listeners by active communication from the brick to the host over the bluetooth connection.

Figure 5 shows the software layers of our architecture. On top is the concrete application model. It uses the F*-API, so classes like *FNXT, FNavigator, FMotor, FSensor* etc. Each F*-Object has a associated adapter instance, which either delegates the calls to the *ICommand API* (Part of the LeJOS package) or simulates the behavior. This is used for testing purposes. The *ICommand API* uses a bluetooth connection and sends simple byte-array commands to the LeJOS firmware on the NXT brick. There the commands are interpreted and mapped to hardware resources.

4. SOFTWARE USAGE AND MODELING

The easiest way to use the Fujaba NXT Library is to start it in debug mode and use eDOBS to create the initial FXNT instance. After that, the methods on the individual motor and sensor instances can be called interactively. Figure 6 shows Eclipse/eDOBS running the library. On the left, you can see the *FNavigator* methods. A call to *travel()* will make the robot drive immediately and block until it finishes.

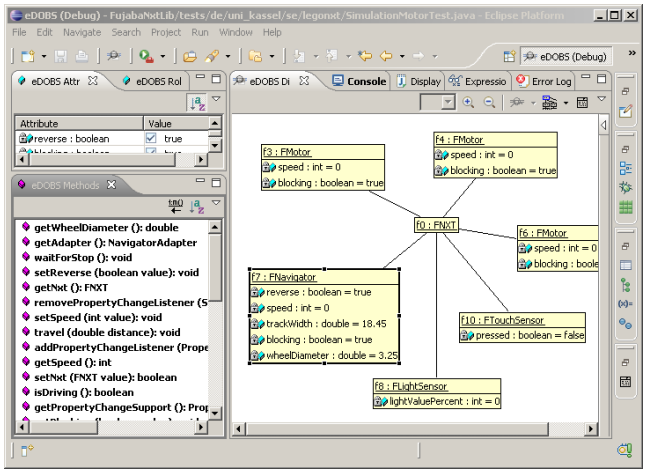


Figure 6: Eclipse/eDOBS running the Fujaba NXT Library

To model the application behavior, both Story or State Diagrams are suitable. To simplify matters, we currently use only story diagrams. Figure 7 shows an example. The method *findInitialPlace()* does not require any control flow, thus it is modeled as just one story activity. However, the single activity itself contains multiple steps and concurrency: some methods (*forward(), backward()*) are asynchronous, that means the method call immediately returns, but the robot acts until a contrary method (*stop()* in that case) is called. After calling a asynchronous method, we use a waitFor-Some-Event-method, in this case *waitForPressed()/ waitForReleased()* on the touch sensor instance. This is a simplification for easy modeling: instead of implementing a listener and being forced to handle the event in some other place of the model, this method blocks until an event of the desired type occurs. This makes reactive programming a bit easier, but semantically means that the single activity is divided into two timing states: the robot drives forward until the sensor hits something, then backwards until the robot is free again.

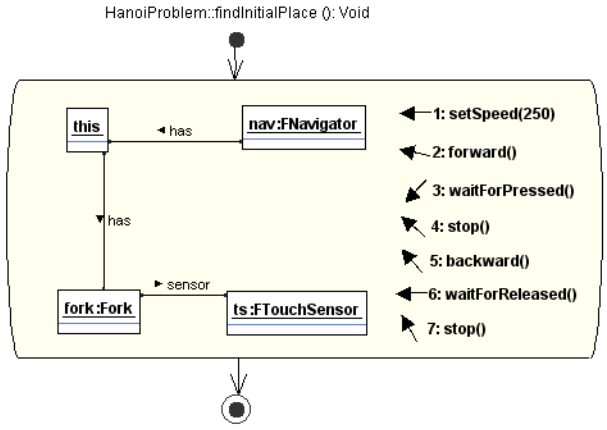


Figure 7: Example of a Hanoi method for finding the blocks

5. PUTTING IT ALL TOGETHER: SOLVING HANOI

Figure 8 shows the class diagram of our Hanoi robot model. It consists of three Fujaba projects: The FujabaNxtLib provides the FNXT, ForkLiftRobot and Fork, thus the abstract modeling to control the hardware. The classes Place, Disc, Hanoi and Solver come from a abstract Hanoi project which can be run or debugged independently from any hardware. The HanoiRobot project, consisting of just HanoiProblem and RobotSolver, depends on the both projects and connects these. We just need to override Solver.transfer() and hook in concrete robot commands to move blocks around there.

Our modeling of the robot-enabled *transfer()* method is just a reference, students will be given only the FujabaNxtLib Library and then should model their own solution.

6. SUMMARY AND FUTURE WORK

We used the new Lego Mindstorms robotics system to implement a forklift robot which can solve the Towers of Hanoi game. The application model was implemented using Fujaba. It is easy to implement the robot control software using standard Fujaba modeling techniques, which makes it suitable for education. By using Fujaba project dependencies, we were able to develop a Library, so modeling control

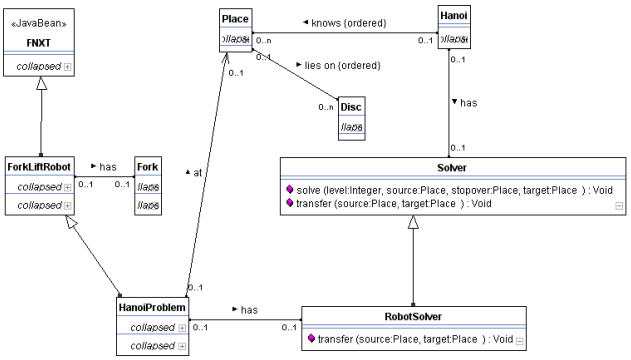


Figure 8: Class diagram of Hanoi, Robot and NXT

software for more applications than just Hanoi is easy. Technically, there are many advantages over the old system: the bluetooth communication is fast and reliable with low latency. On the hardware side, the improved motors with step counters make programming easy as we don't need sensors to detect the motor state anymore. This, and the sensitive fork mechanic free up sensor ports, so the robot can be easily extended with more functionality. We were surprised that the navigation of the robot is quite exact, turning and moving the robot has just a minimal deviation.

We plan to model the forklift as 3D model, so we can simulate it's behavior in a physical environment in software. This should reduce the time for developing and testing software. Due the adapter layer in our software architecture, application models should be able to run in a simulation environment without the need for any changes.

Furthermore, we plan to adapt the Fujaba code generation [3]. This requires a set of templates which consider the limited capabilities of the LeJOS VM, and a runtime library for sensor event handling.

7. REFERENCES

[1] 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*, October 2002.

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

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

[4] LeJOS, Lego Mindstorms NXT. http://mindstorms.lego.com/, 2008.

[5] LeJOS, Java for Lego Mindstorms. http://lejos.sourceforge.net/, 2008.

Experiences with 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

Bernhard Westfechtel
Angewandte Informatik 1
Universität Bayreuth
D-95440 Bayreuth
bernhard.westfechtel@uni-bayreuth.de

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 a recently launched project dedicated to a model-driven modular SCM system confirm this claim. This paper investigates modeling in the large support provided by the object-oriented CASE tool Fujaba and discusses potential extensions of Fujaba based on UML package diagrams.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software architectures—*languages*; D.2.11 [**Software Engineering**]: Management—*software configuration management*

Keywords

version control, packages, imports

1. INTRODUCTION

Software configuration management (SCM) is the discipline of controlling the evolution of large and complex software systems. A wide variety of SCM tools and systems has been implemented, ranging from small tools such as RCS [10] over medium-sized systems such as CVS [11] or Subversion [4] to large-scale industrial systems such as ClearCase [12].

Version control is a core function of any SCM system. Version control is based on *version models*, many of which have been implemented in research prototypes, open source products, and commercial systems [5]. While there are considerable differences among these version models, it is also true that similar concepts such as revisions, variants, state- and change-based versioning appear over and over again. Unfortunately, version models are usually implicitly contained in implemented systems.

Thus, the SCM domain is characterized by a large number of systems with more or less similar features incorporating hard-wired version models which have been implemented with considerable effort. This observation has motivated us to set up a project dedicated to a *model-driven modular SCM system* (abbreviated as *MOD2-SCM* [2]):

First, *version models* are defined *explicitly* rather than implicitly in the code. This makes it easier to communicate and reason about version models. Second, modeling comprises both *structure* and *behavior*. Furthermore, behavioral models are executable. Third, productivity is improved by replacing programming with the creation of *executable models*. Fourth, version models are not created from scratch. Rather, reuse is performed on the modeling level by following a *product line* approach [3]. Finally, the product line is based on a *model library* which is composed of reusable and loosely coupled architectural units.

In MOD2-SCM, we decided to use the object-oriented modeling language and environment *Fujaba* [15] because it supports generation of executable (Java) code from a UML model. To date, only a few approaches have been dedicated to *model-driven development of versioning systems* [14, 13, 7]. However, all of these approaches are confined to structural models inasmuch as the behavior is hard-coded into the respective system.

In this paper, we investigate *modeling in the large* with and beyond Fujaba. As to be demonstrated, the model currently being developed in the MOD2-SCM project is fairly large. Furthermore, the success of the project heavily depends on a carefully designed *model architecture* with loosely coupled architectural units [2]: The product line should support a set of variation points which may be combined in an orthogonal way as far as possible. To this end, the coupling between architectural units has to be reduced to a minimum.

2. MODELING IN THE LARGE

In object-oriented modeling, modeling in the large is an area which has not attracted sufficient attention so far. In the following, we will first discuss support for modeling in the large as far as it is provided in the current version of Fujaba. Subsequently, we will show how external tool support may be used to complement the functionality of Fujaba by creating package diagrams from generated Java code. Finally, we will discuss package diagrams in UML 2.0.

2.1 Support in Fujaba 5.1

On a coarse-grained level, Fujaba models are organized into *projects*. A project stores a model in a single file. A model stored as a project may be self-contained, or it may reference models stored in external projects. These references imply *dependencies* between projects. In this way, both physical decomposition and model reuse are supported.

Within a project, model elements are created in diagrams. *Class diagrams* serve as the primary means for structuring. When a class is created, it may be assigned to a *package* in the dialog window for class editing. Fujaba maintains a tree view of diagrams and model elements, but packages can be neither defined nor displayed in the tree view. Furthermore, Fujaba does not support *package diagrams*.

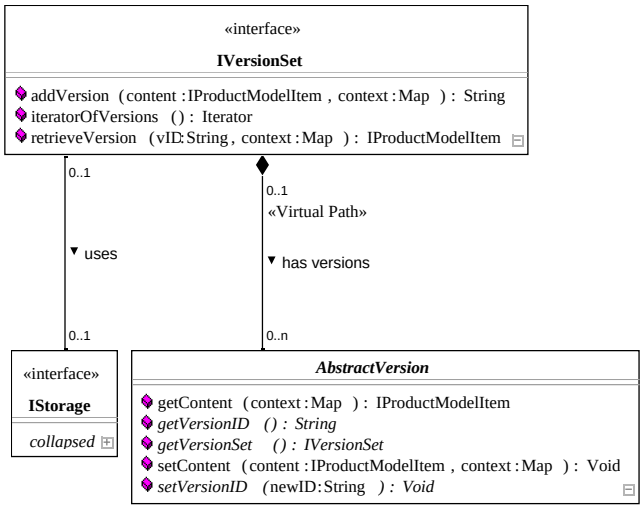


Figure 1: Excerpt of a Fujaba class diagram

In a class diagram, the owning package of a class is not displayed. Furthermore, the class diagram itself is not owned by a package. As a consequence, there is no distinction between references and declarations of classes, and it is not obvious from which packages the classes originate.

To some extent, conventions may be used to structure the model. Such conventions may state e.g. that exactly one class diagram is introduced for each package, the package name is used to identify the class diagram, and all external classes are displayed with collapsed attribute and method sections. The latter is demonstrated in Figure 1, which shows an excerpt of the class diagram for maintaining version sets (package core.versions, see later). Interface IStorage was imported from another package.

2.2 Reverse Engineering with eUML

To complement the modeling facilities of Fujaba, we used the *eUML* plugin of Eclipse to generate a package diagram from the Java code created by the Fujaba compiler. The package diagram for the model in its current state of evolution is displayed in Figure 2. eUML distinguishes between different kinds of dependencies, resulting e.g. from imports, specializations, instantiations, and method calls in the Java code. The eUML user may configure the kinds of dependencies displayed in the diagram.

While the eUML package diagram is helpful, it still suffers from several limitations. First, since it is reverse engineered from the generated Java code, it cannot be used for designing the model architecture up front. Second, the resulting graph is rather dense since it also includes "secondary" dependencies (e.g., to call a method, the class of a parameter may have to be imported, as well). Third, the diagram displays implementation-level dependencies, i.e., dependencies in the generated code, which may differ from conceptual dependencies.

Let us give an example for the latter: In Figure 1, a version set (interface IVersionSet of package core.versions) makes use of a storage (interface IStorage of package core.storage). This association is introduced in the package core.versions: The storage stores a set of versions using deltas, but it is independent of the way how the version set is organized on a conceptual level. On a conceptual level, core.versions depends on core.storage but not vice versa. However, for a bidirectional association Fujaba generates methods for navigation at both ends, introducing cyclic dependencies in the generated code[1].

2.3 UML 2.0 Package Diagrams

Let us briefly recall the concepts which UML 2.0 offers for structuring large models [8, 9]: 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 full 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. Apart from nesting and imports, UML 2.0 offers *package merges*, which will not be discussed in this paper.

Figure 3 shows a UML package diagram for the current MOD2-SCM model. Please note the differences to the eUML diagram of Figure 2: First, the UML diagram visualizes nesting of packages. Second, only conceptual relationships are shown (e.g., core.versions imports core.storage but not vice versa). Finally, the number of relationships is significantly reduced due to the transitivity of public imports.

There are fundamental differences between imports in UML 2.0 and imports in *modular programming languages* such as Modula-2 and Ada. In these languages, each — separately compiled — program unit (called module in Modula-2 and package in Ada) may reference only its own local declarations unless the namespace is extended by an import. Depending on the kind of import, imported elements may be referenced with or without qualification. Furthermore, an imported element may only be *used* but not modified. In contrast, the UML 2.0 standard states ([8], p. 143): "An element import ... works by reference, which means that it is not possible to add features to the element import itself, but it is possible to modify the referenced element in the namespace from which it was imported." For example, an imported class may be inserted into a class diagram of the importing package and extended with attributes, methods, and associations; these extensions apply to the imported package where the imported class is declared.

Altogether, in UML 2.0 the rules for referencing and modifying elements are liberal and make it convenient to access external elements. Unfortunately, these rules may threaten the *modularity* of a UML model. In the case of an undisciplined modeling process, it is fairly easy to create an archi-

[1]While we could have used a unidirectional association in this example, it would be far too restrictive to enforce unidirectional cross-package associations in general.

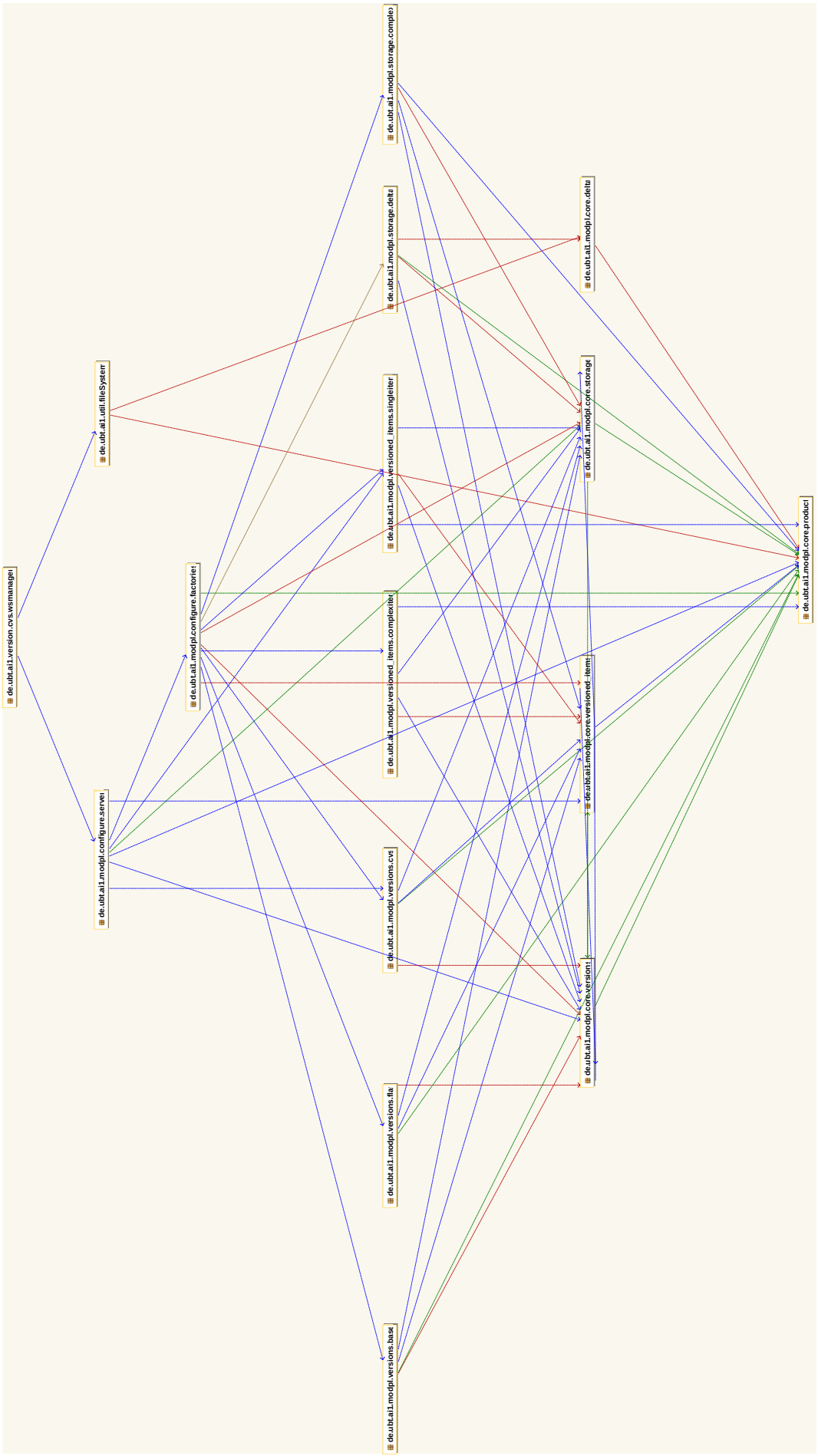


Figure 2: eUML package diagram extracted from generated Java code

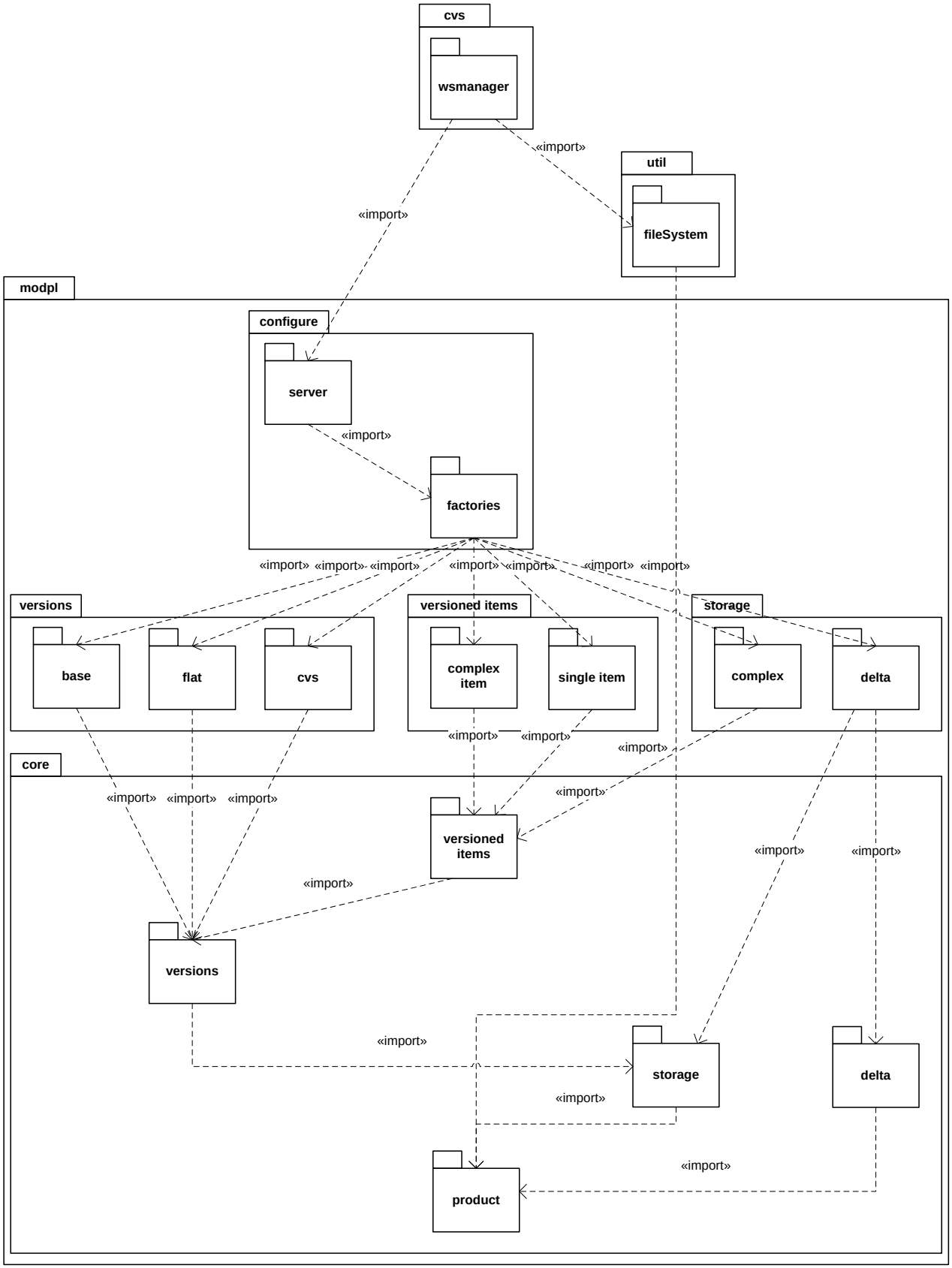


Figure 3: UML package diagram using nesting and public imports

tecture with tightly coupled packages — which would violate the goals we pursue in the MOD2-SCM project.

In particular, there is no guarantee that a package diagram such as shown in Figure 3 shows the actual dependencies between packages in the architecture. First, it is possible to reference external elements without imports by using fully qualified names; the implied dependencies would not be displayed in a package diagram showing only import relationships. Second, public imports are transitive. Thus, when importing some package, all packages of the transitive closure of public imports are visible, as well. A package diagram with public imports does not tell which of these packages are actually referenced.

For example, in Figure 3 the package configure.server imports configure.factories and thus may reference all packages below. If the actual dependencies were so comprehensive, we would have to be concerned about the modularity of the system. Figure 2 shows that only a few dependencies emanate from the server package. Still, there is one dependency on the package versions.cvs which appears to be suspicious: Why should a configurable server depend on a specific version model such as the CVS model?

This example demonstrates that the package diagram with public imports appears to be elegant, but is too imprecise: There is no way around inspecting all actual dependencies emanating from a package. To support such analyses, private imports would be more useful: As Java imports, private imports are not transitive, forcing the client to explicitly import all packages on which it depends. Of course, private imports imply a much denser diagram similar to the one shown in Figure 2.

3. CONCLUSION

In this paper, we reported on experiences with modeling in the large in Fujaba, referring to a recently launched project for developing a model-driven product line for SCM systems. Our experiences indicate that reverse engineering of the model architecture with an external tool is not sufficient and thus improved support for modeling in the large in Fujaba itself is urgently needed. We also discussed UML 2.0 package diagrams as a notation for model architectures, focusing on package imports (the fairly complex concept of package merge goes beyond the scope of this paper, see e.g. [6]). Please note that package diagrams are available in MOFLON [1], which has been built on top of Fujaba. However, MOFLON currently supports only public imports, while our experiences demonstrate that private imports are needed, as well.

To conclude, we give a few suggestions for extending Fujaba with support for modeling in the large: First, the tree view should be revised such that it allows to define a package hierarchy. Second, not only model elements such as classes and associations, but also class and story diagrams should be assigned uniquely to one package. Third, a graphical editor for package diagrams should be offered (supporting nesting of packages as well as public and private imports). Fourth, model elements may be referenced only where they are visible. In addition, we recommend to implement some restrictions which deviate from the UML 2.0 standard: First, qualified references should be disallowed to avoid hidden dependencies. Second, it should be prohibited to modify imported elements.

4. REFERENCES

[1] 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 4066 of *LNCS*, pages 361–375, Heidelberg, 2006. Springer.

[2] T. Buchmann, A. Dotor, and B. Westfechtel. MOD2-SCM: Experiences with co-evolving models when designing a modular SCM system. In *Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management*, Toulouse, France, Oct. 2008.

[3] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, Boston, Massachusetts, 2005.

[4] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion*. O'Reilly & Associates, Sebastopol, California, 2004.

[5] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[6] J. Dingel, Z. Diskin, and A. Zito. Understanding and improving UML package merge. *Software and Systems Modeling*, Dec. 2007. Online First.

[7] J. Kovše. *Model-Driven Development of Versioning Systems*. PhD thesis, University of Kaiserslautern, Kaiserslautern, Germany, Aug. 2005.

[8] Object Management Group, Needham, Massachusetts. *OMG Unified Modeling Language (OMG UML), Infrastructure, V 2.1.2*, formal/2007-11-04 edition, Nov. 2007.

[9] Object Management Group, Needham, Massachusetts. *OMG Unified Modeling Language (OMG UML), Superstructure, V 2.1.2*, formal/2007-11-02 edition, Nov. 2007.

[10] W. F. Tichy. RCS – A system for version control. *Software: Practice and Experience*, 15(7):637–654, July 1985.

[11] J. Vesperman. *Essential CVS*. O'Reilly & Associates, Sebastopol, California, 2006.

[12] B. A. White. *Software Configuration Management Strategies and Rational ClearCase*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, 2003.

[13] E. J. Whitehead, G. Ge, and K. Pan. Automatic generation of hypertext system repositories: a model driven approach. In *15th ACM Conference on Hypertext and Hypermedia*, pages 205–214, Santa Cruz, CA, Aug. 2004. ACM Press.

[14] E. J. Whitehead and D. Gordon. Uniform comparison of configuration management data models. In B. Westfechtel and A. van der Hoek, editors, *Software Configuration Management: ICSE Workshops SCM 2001 and SCM 2003*, volume 2649 of *LNCS*, pages 70–85. Springer, 2003.

[15] A. Zündorf. Rigorous object oriented software development. Technical report, University of Paderborn, Germany, 2001.

Fujaba goes Web 2.0

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

ABSTRACT

The latest research activities of the Fujaba group of Kassel University led to challenges in development of new web technologies enabling end users to wrap services into web gadgets and to combine them into complex web applications. Web applications running inside a webbrowser bring new requirements to the traditional desktop application development process. Since web applications usually don't come to life using model based approaches or Story Driven Modelling the Fujaba Toolsuite has to be adopted to fulfil the new requirements.

The Fujaba group of Kassel wanted to get all the help we are used to get from Fujaba in ordinary application development for our web applications, too. Thus, we are developing a new code generation for the generation of Google Web Toolkit compliant Java code that will then be compiled into JavaScript running on web browsers. In addition, we develop tool support for building the GUIs of such web applications. And, on the server side, we develop technologies to facilitate service development with Fujaba. This paper reports on the design of these new Fujaba web components.

1. INTRODUCTION

Since March 2008 the Fujaba group at Kassel University participates in the European Project FAST: Fast and Advanced Storyboard Tools. This project aims at the development of a new visual programming language and tools that will facilitate the development of complex front-end gadgets. A gadget in this opinion is a small web application that is designed to run inside a so called mashup platform. Some of these platforms like Yahoo Pipes[9], iGoogle[6] or Microsofts Popfly[8] are widely used in the present. Since one of the project partners has already devolped a mashup platform, the so called EzWeb platform [3], the Fujaba team at Kassel tries to support gadgets for this platform at first and focuses its research on the specific needs of EzWeb web gadgets. Since Kassel sees gadgets as kind of web application, the research regarding Fujaba and the development of web applications with Fujaba is not only focused on gadgets needed for FAST, but on what we call Web 2.0 applications in general. These applications and their development differ from the standard desktop applications known to be generated with Fujaba. There are many challenges that have to be addressed to get applications running over the web in the way we want them to. The main problem is the lack of possibilities you have on the client. Since the applications are intended to run inside a web browser the client side has to be JavaScript. The old way of web applications were web formulars on client side which communicate with the server and are completely blocked while waiting for server responses. Web 2.0 applications in our manner will be Ajax applications, meaning that the whole communication between client and server will be asynchronous giving the user possibilities to interact with the application even when waiting for server responses. Another problem you have in the development of web applications is the distribution of data, or more specifically replication of data models between the server and one or multiple clients. In the domain of web applications developing client side code is done by hand coding JavaScript for the browser. The Fujaba team at Kassel wanted to have all the opportunities known from standard software development processes: code generation, model based development and story driven modelling to benefit the development of web applications and gadgets. Thus, we chose to adapt Fujaba and its tools to suffice the new needs.

Figure 1 shows the architecture of what we call Web 2.0 application. On the client side a web browser may show some web gadgets. We propose to program these web gadgets as model view controller patterns, i.e. to separate the representation and the data model. The representation may employ some domain object model of the web page shown to the user. Note, our reference architecture for Web 2.0 applications is somewhat unconventional. Usually, the client has only view data and the server has the model data and the application logic. In our reference architecture, the application logic, i.e the business rules and model transformations are to a large extent executed on the web clients based on the model data available on that client. For GUI operations we develop a dedicated version of Fujaba Story Diagrams that will provide DOM rewrite rules within concrete syntax, cf. section 2. The application logic is based on an application data model. This application data model and its operation may be developed with usual Fujaba story diagrams. In order to deploy the resulting Java code on a web browser, we will use Google Web Toolkit technology [4], cf. section 3. Synchronization between DOM and application data model is done via the usual property change based event mechanisms.

For the development of (web) services we will use standard Story Driven Modeling technologies. Note, some service models and some application models may overlap. In these cases, we develop the data model only once and deploy it on client and on server side, equally. For certain applications it may suffice to copy (parts of) the server data to the client where the data is visualized and exploited using the

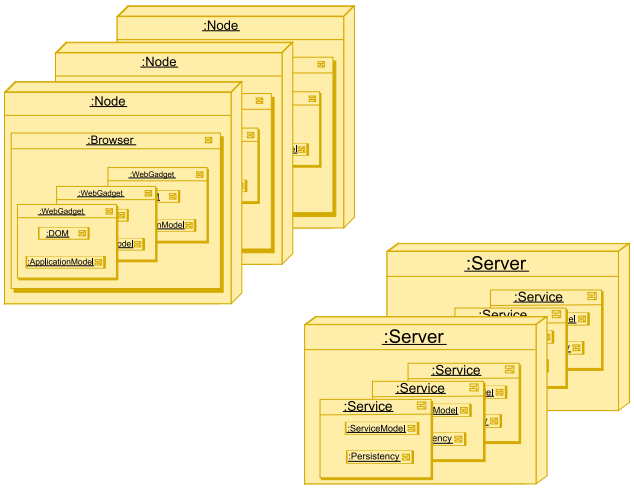
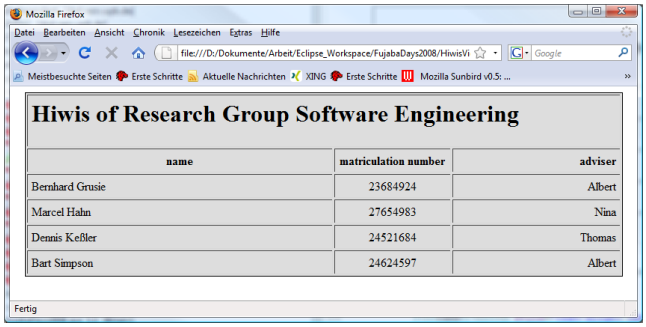


Figure 1: Reference Architecture for Web 2.0 Applications

user interface mechanisms described above. Then, modified data may be copied back to the server in order to make it persistent or in order to share it with other client instances. Therefore, we will provide data replication mechanisms and persistency mechanisms described in section 2.

Next, the problem of service orchestration is addressed by exploiting new Web 2.0 mashup technologies and by a dedicated work flow support service that we develop within the FAST project, cf. section 5.

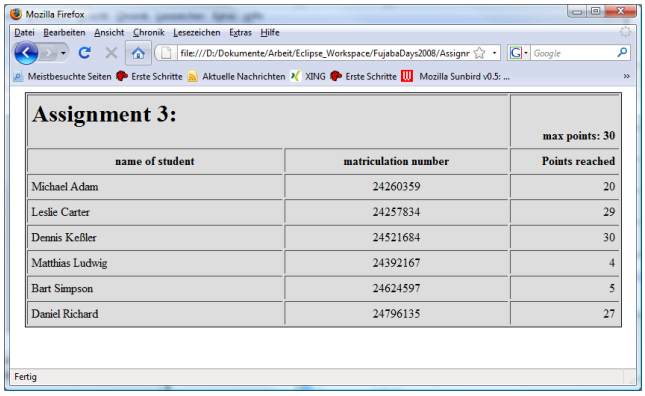


Hiwis of Research Group Software Engineering

name	matriculation number	adviser
Bernhard Grusie	23684924	Albert
Marcel Hahn	27654983	Nina
Dennis Keßler	24521684	Thomas
Bart Simpson	24624597	Albert

Figure 2: Tutor Management Application

As running example of our approach we modeled a web enabled homework management system for students and tutors. As first feature, the homework management system allows to manage courses, students and tutors. Only the lecturer staff is allowed to do so. This is shown in Figure 2. As second feature, the homework management system supports the grading of students' homeworks. As shown in Figure 3, we award the student homework with points. A certain amount of points are required to pass the course. Security requirements have to be considered to implement that feature, however this is not in our focus at this stage of work. As third feature, it should be possible for students to submit their homework directly via web interface in our system, for example by filling out multiple choice questionnaires, entering answer text or even uploading programming source code. The submitted homeworks should be reviewed



Assignment 3: max points: 30

name of student	matriculation number	Points reached
Michael Adam	24260359	20
Leslie Carter	24257834	29
Dennis Keßler	24521684	30
Matthias Ludwig	24392167	4
Bart Simpson	24624597	5
Daniel Richard	24796135	27

Figure 3: Student Assignments Application

and corrected by our tutors. After correcting a homework, the tutor enters the results directly into the system, and students can view their homework results.

Not all of the above requirements need to be fulfilled for this example. It is sufficient to hold the data model on the server, for example. The homework management system in the design described above is more similar to traditional web applications, even if the communication between client and server will be asynchronous. For development of the Fujaba tools needed for new Web 2.0 applications it is a good example, because it will lead to less complex client side code. It will need persistency of the data, user management, multi-user capabilitiy and GUI operations, so we can test new Fujaba tools with it. For more complex examples, we will go for the ToDo list gadget developed in terms of the FAST project, cf. 5 which will then be inteded to run inside the EzWeb platform and which will have a client side data model, on top.

2. GUI AND DOM OPERATIONS

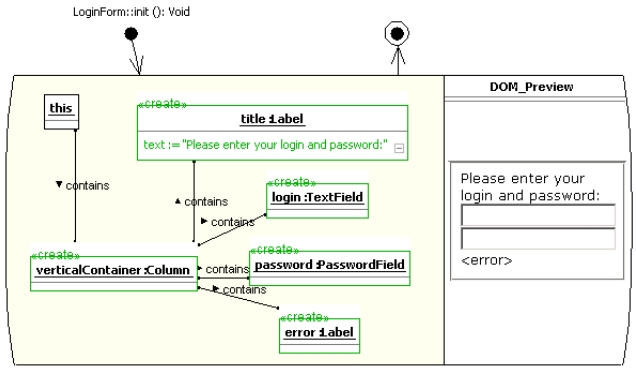


Figure 4: DOM Modeling in Fujaba

The Graphical User Interface (GUI) of our application is realized using up-to-date AJAX (Asynchronous JavaScript and XML) technologies. In oder to avoid any direct JavaScript programming, we use the Google Web Toolkit. It provides a Java API with GUI Widgets as abstraction of HTML, CSS and JavaScript browser technologies. Thus, creating and modifying the GUI are just manipulations in a Java object structure representing the DOM (Document Object Model)

in the client browser. The GWT maps those operations to JavaScript calls modifying the HTML document tree. Traditionally, Fujaba Story Diagrams are used to graphically model application behaviour [1], [10]. We are currently trying to adopt these diagrams to also model the GUI in Fujaba. Figure 4 shows an example of specifying a simple login form in Fujaba, along with the concrete syntax preview on the right side.

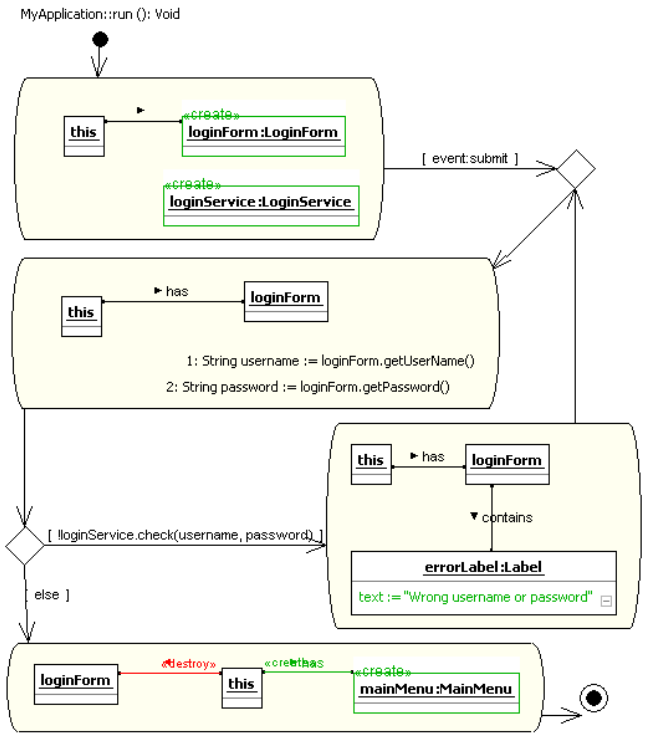


Figure 5: Application Flow Modeling in Fujaba

Changes to the current GUI (layout and data) will be usually performed when a user action occurs. This can be a click on a button, entering text or any other mouse- or keyboard related event. These events will be handled locally in the client. To model the client behavior, we plan to extend the Story Diagram or State Diagram transitions to react on user input events. Figure 5 shows the modeling of such a GUI state change: Initially, the application display a login form. When pressing enter in the text fields, which is mapped to the *submit* event, the application state switches to the main menu or displays an error message.

3. CLIENT SIDE APPLICATION MODEL

For our example assignment management application, the application model is pretty simple, cf. Figure 6.

This application model is specified with Fujaba. Then, we generate Java code for the implementation of the described data. This implementation shall be used to represent runtime data on the client side, i.e. within the client's web browser. Therefore, we use Google's Web Toolkit (GWT) [4] to compile the generated Java code into JavaScript code that is deployed on the web client. Using GWT has the advantage that the template based code generation of Fujaba needs only minimal adaption. Actually, we only need to

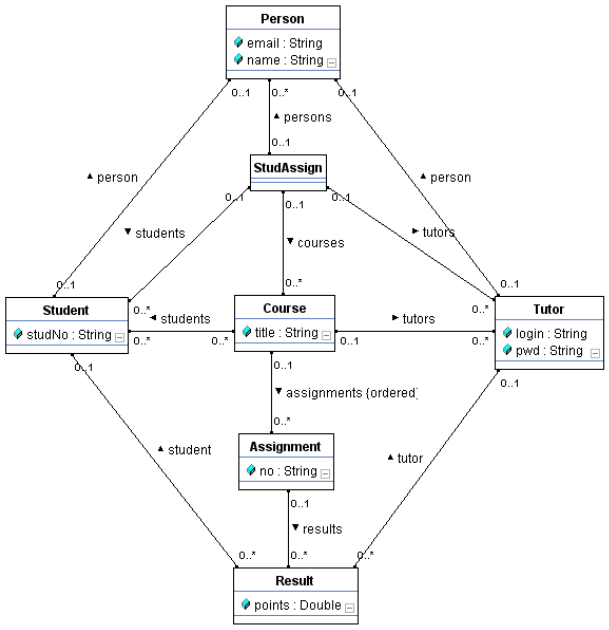


Figure 6: Assignment Management Model

adapt the container classes deployed for the implementation of to-many associations. To implement these associations data structures are needed to manage a set of neighbouring objects. GWT supports the standard Set implementations of the Java Collection API which does not support concurrent modification of its contents. Therefore, Fujaba traditionally uses its own Collection classes. Since GWT is not able to compile this and other Fujaba specific classes into JavaScript we will need to use GWT containers where they are needed. The only thing to do for this is providing new templates for the code generation. In addition to the Java-to-JavaScript compilation, the GWT approach has the advantage, that a special GWT browser allows debugging of client side code. Thus, with the usage of GWT, we will be able to generate code from Story Diagrams including graph rewrite rules and to do design level debugging of the execution of such rewrite rules for web clients.

To enable the model view controller pattern for connecting the clients DOM based GUI with the clients application data model, we need a property change management infrastructure. In [2] the CoObRA framework was presented as a generic property change and replication mechanism which we now have adapted to the specific needs of web application development. Thus, there will be a CoObRA component on our web clients that provides undo redo for the web application and that may be used to send change protocols from the clients to the services. The latter mechanism enables CoObRA based data replication mechanism and team collaborations. Thus, in our example application, multiple tutors may work with the assignment management application concurrently. Each tutor may enter the grades for his or her students and these changes are then replicated / committed to the assignment management service where this data is then replicated to other tutors. Thus, each tutor sees the progress of the overall work and whether some student has already been graded for a certain assignment by some team

mate.

The infrastructure for property changes and for CoObRA like change protocols are current work.

4. SERVICE DEVELOPMENT

On the server side, we deploy a server data model and a persistency component. In simple cases the server data model is the same data model as the application data model. This will e.g. hold for our assignment management service and for our tutor management service. This data model is generated from the same Fujaba model that is used for the generation of the application data model. As can be seen from our reference architecture, the client can have application logic and data model, too. However, in some applications, the client may load only a certain part of the whole data, e.g. for performance or security reasons. In this case some operations that require access to the whole data storage may be deployed on server side. In addition, the server needs to manage which clients are currently online and which client has loaded which part of the overall data. In case of a data update the server has to compute which clients have replicas of the corresponding data fragments and thus require an update notification.

In general, the server component needs to provide some support for the access protocols used to communicate with the web clients e.g. a simple object access mechanism.

In addition, the server needs a persistency component in order to store data e.g. for recovery purposes. Here we use the standard CoObRA mechanisms. Note, these CoObRA mechanisms allow direct storage of each change for recovery as well as a transaction concept grouping change sets for consistency purpose as well as a cvs like update / commit usage.

Using these mechanisms, the generation of simple (data replication) services becomes quite simple. Just generate it from the Fujaba model. However, note that such services have a certain scaling problem. CoObRA based applications / services keep the whole runtime data in main memory and are thus restricted to data model sizes that fit into 2 gigabyte main memory on 32 bit Java virtual machines. This should not easily become a problem for our assignment service or for the tutor service. However, one would not run a banking service or a web shop on this limited data size. To overcome this scaling problem, we think of a combination of a CoObRA mechanism coordinating the data replication and a relational database for management of large data volumes. This is future work.

5. SERVICE ORCHESTRATION

As mentioned before, within the FAST EU project the University of Kassel takes part in the development of tools to support complex web gadgets. In our own research we try to develop a ToDo list gadget with the help of the Fujaba adoptions described in this paper. The ToDo list gadget may be used to organize multiple services, cf. Figure 7.

The idea is that a number of steps is defined where each step refers to some service via an URL. This allows to visit one service after the other. Next, each step may be equipped with named input and output data ports, e.g. for the departure and destination airport for some flight booking service. Via such data ports, the mashup platform may retrieve data as e.g. the flight arrival time from one service and forward

Have a meeting

No.	Action Item	State	Documents	Responsible	Input Data	Output Data
1	Book flights	☑	http://www.expedia.de/fluege/linienfluege	Nina	dates, airport of departure, destination airport	e-tickets, departure times, arrival times
2	Book hotel	☑	http://www.expedia.de/hotels/default	Ruben	dates, location, estimated price	Booking confirmation
3	Have presentations	☑	http://docs.google.com/Doc?docid=xxxxxxxxxxxx3	Albert	work done, ideas	-
4	Have social dinner	☑	-	UPM	date, time, number of persons	a lot of fun
5	Reimbursement for travel expenses	☐	http://www.uni-kassel.de/pvabt3/download/reisekosten.ghk	Nina, Ruben, Albert	dates, bills	money

Figure 7: ToDo List Gadget for Service Orchestration

it to another service. Together with some means for the automation of todo item execution, this may help to organize and to orchestrate multiple services within a common work flow. The emerging software can be run either as stand alone web application, or inside the mashup platform. Combining client side application model, multi user execution with data replication, automated todo item execution and data persistency the web based workflow management system can be seen as complex Web 2.0 application nearly entirely developed with Fujaba. One major step with this application will be the deployment inside EzWeb as mashup platform, since it is not used, maybe not even intended for gadgets as complex as our workflow example. The main difference between traditional gadgets and ours is not only its complexity, but also the fact that the workflow management system not only uses web services inside the todo item, it needs is own services in the background to establish persistency and data replication via the CoObRA mechanism. These new ideas have to be carafully transported to EzWeb together with possibilites which enable the end-user to wrap the services they want to use inside their workflow, which means producing tools to help the user to define in and out ports of the todo item in an easy way. For this purpose the Kassel University team currently tries to develop so called Clipboards to enable point and click mechanisms to do this. These technologies are in a very early stage and will be issued in detail in future publications.

6. SUMMARY AND FUTURE WORK

Honestly, most of the described ideas are current and future work. Currently, we have a prototype of a ToDo List gadget that has been build according to the reference architecture. Because the GUI building mechanisms are not yet ready to be used, we built the user interface by hand. The data replication and persistency were at first built upon a foreign GWT library called jstm4gwt [7]. Many parts of this library had to be adopted, because the property change mechnisms didn't work as expected and so we have adopted the CoObRA mechanisms for GWT earlier than expected to get rid of this library. Currently we are moving our application from the jstm4gwt version to a new one that will use

the CoObRA mechanism. Nevertheless, we could already prove that web based workflow handling is possible with the jstm4gwt version.

To get the GUI development more comfortable, besides the enhanced Fujaba Story Diagrams we plan to develop a GUI designer which will support standard GWT and GWT-Ext [5] Widgets for the programming of user interfaces. The GUI designer is planned as plugin for Fujaba4Eclipse and will enable the user to graphically design the GUI. As a reference we will try to use the former Eclipse plugin Visual Editor, which was able to built SWT and Swing GUIs inside Eclipse in a visual way. Out of the graphical representation the sourcecode was generated. The Fujaba GWT GUI designer will not generate sourcecode directly out of the graphical Widget representations, it will generate the appropriate classes and objects that will be displayed in Fujaba classdiagrams and Story Diagrams.

For further enhancement of the Web 2.0 application development process we plan to extend Fujaba in a way to automatically generate the GWT Module definition, entry class and .html host website when a web application is the intended target. Solving the problems stated in the introduction with well known software engineering principles will yield new tools and frameworks in the Fujaba Toolsuite context raising the software development process for web applications to a model centric level, yet unknown.

7. REFERENCES

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

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

[3] EzWeb. http://ezweb.morfeo-project.org/, 2008.

[4] The Google Web Toolkit. http://code.google.com/webtoolkit, 2008.

[5] GWT-Ext. http://code.google.com/p/gwt-ext/, 2008.

[6] iGoogle. http://www.google.com/ig, 2008.

[7] XSTM. http://www.xstm.net/, 2008.

[8] Microsoft popfly. http://www.popfly.com/, 2008.

[9] Yahoo Pipes. http://pipes.yahoo.com/pipes/, 2008.

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

Towards a Hybrid Transformation Language: Implicit and Explicit Rule Scheduling in Story Diagrams

Bart Meyers
University of Antwerp
Antwerpen, Belgium
bart.meyers@student.ua.ac.be

Pieter Van Gorp
University of Antwerp
Antwerpen, Belgium
pieter.vangorp@ua.ac.be

ABSTRACT

Transformation rules can be controlled explicitly using language constructs such as a loop or a conditional. This approach is realized in Fujaba's Story Diagrams, in VMTS, MOLA and Progres. Alternatively, transformation rules can be controlled implicitly using a fixed strategy. This approach is realized in AGG and AToM^3. When modeling transformation systems using one approach exclusively, particular aspects could have been expressed more intuitively using the other approach. Unfortunately, most (if not all) transformation languages do not enable one to model the control of some rules explicitly while leaving the control of other rules unspecified. Therefore, this paper proposes the extension of Story Diagrams with support for implicit rule scheduling. By relying on a UML profile and on higher order transformations, the language construct is not only executable on the MoTMoT tool, but on any tool that supports the standard UML syntax for Fujaba's Story Diagrams.

Keywords

Transformation Languages, Rule Scheduling, Higher Order Transformations, Language Engineering

1. INTRODUCTION

Transformations are critical in model-driven development. Since homemade modeling languages may be defined and integrated at any time in the development process of a software system, transformations have to be tailored accordingly to integrate these new languages before they can actually be used. Therefore, a highly expressive transformation language is very useful, because it facilitates defining the needed transformations.

When defining transformation languages, certain choices are made. The language can be imperative (i.e., operational) or declarative [8, Chapter 2]. Explicit rule scheduling mechanisms (e.g., conditionals) tend to be called *imperative* since they enable one to model the execution of transformation rules in terms of the *state* of the transformation system. Languages with implicit rule scheduling (such as AGG) tend to be called *declarative* due to the absence of an explicit state concept. There is no better choice in this matter. For certain problems implicit rule scheduling feels more intuitive, sometimes explicit rule scheduling turns out to be convenient. Therefore, this paper introduces a language construct for the integration of implicit rule scheduling in an imperative language. The integrated language is *hybrid* (imperative as well as declarative) with regards to rule scheduling.

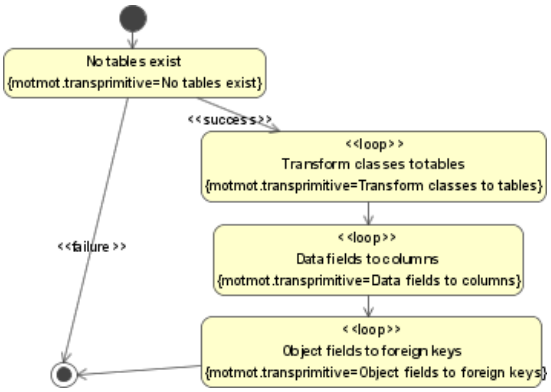


Figure 1: Control flow of a transformation from class diagrams to a relational database schema.

The remainder of this paper consists of the following: Section 2 motivates the need for the hybrid transformation language by means of an example. Section 3 describes a higher order transformation for mapping hybrid transformation models to fully imperative transformation models in ordinary Story Diagrams. This defines a compiler for the new language construct. Section 4 presents related work and Section 5 discusses future work. Finally, Section 6 summarizes with a conclusion.

2. IMPLICIT RULE SCHEDULING

Consider the example of Figure 1, which represents a simplified model transformation from class diagrams to relational database schemata. This example has been used for comparing transformation languages before [1]. Figure 1 displays a story diagram that applies explicit rule scheduling exclusively. Throughout this paper, the UML profile for story diagrams is used. The benefits of using a profile rather than metamodels are discussed in Schippers et al. [6].

In the UML profile, rules (annotated class diagrams) are embedded in a control flow (annotated activity diagrams) by using a tag named *transprimitive* whose value points to the UML package containing the transformation rule (i.e., classes and associations annotated with $\ll create \gg$, $\ll destroy \gg$, etc. [8]). In effect, the value of the *transprimitive* tag states which rewrite rule needs to be executed when the transformation system is in a particular state. Fujaba realizes the same semantics but differs syntactically by visually embedding the rewrite rules instead of referring to their name.

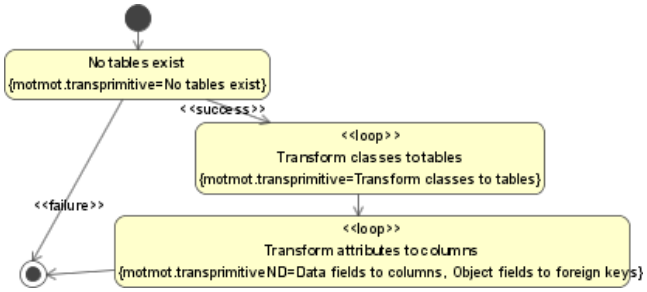


Figure 2: Example of the usage of the new language construct.

In Figure 1, the first rule that has to be executed checks whether there are tables in the database. If some tables exist in the database, the algorithm ends since otherwise existing data may be corrupted by the simple transformation. If not, the actual creation of schema elements is started: first, classes are transformed to tables. Next, class attributes must be transformed. There are two types of attributes: object references and primitive values. It turns out that the transformation of the two kinds of attributes can be modeled elegantly with two transformation rules. Attributes of a simple data type `t` become columns of type `t`. Object attributes of class `T` become columns of type integer containing key references to the primary unique ID column of table `T`. Additionally, a foreign key constraint is added to the database. Therefore, in Figure 1, all data fields are transformed, and then all object fields are transformed.

Having to express the transformation of attributes in two sequentially executed rules decreases the quality of the transformation model in several ways. First, one has to impose an order on these two rules, which is useless and has no meaning. This is a clear case of over-specification. Secondly, the transformation of all attributes is conceptually one action, and should be modeled as such.

Alternatively, one could have modeled the transformation using implicit rule scheduling. However, in that case, the "no tables exist" test (this kind of sanity checks are rather common at the start of model transformations) could not have been scheduled before the other rules without having to rely on hand-written code or other tool-specific approaches. This is a reason why modeling in a language using explicit rule scheduling is a good choice for this problem.

2.1 A new language construct

It turns out that both implicit and explicit rule scheduling are needed to model the example of Figure 1 in a decent way. Therefore, this paper introduces a new language construct for story diagrams that allows implicit rule scheduling. Consider Figure 2 as an example of the usage of this new construct. Analogue to the *transprimitive* tag definition, a new UML tag definition *transprimitiveND* is proposed that can be used for the state that transforms attributes. However, a *transprimitiveND* state can reference more than one UML Package and chooses non-deterministically in which order the packages are executed, hence "ND" in the name.

More general, consider a set of rules that can be executed in

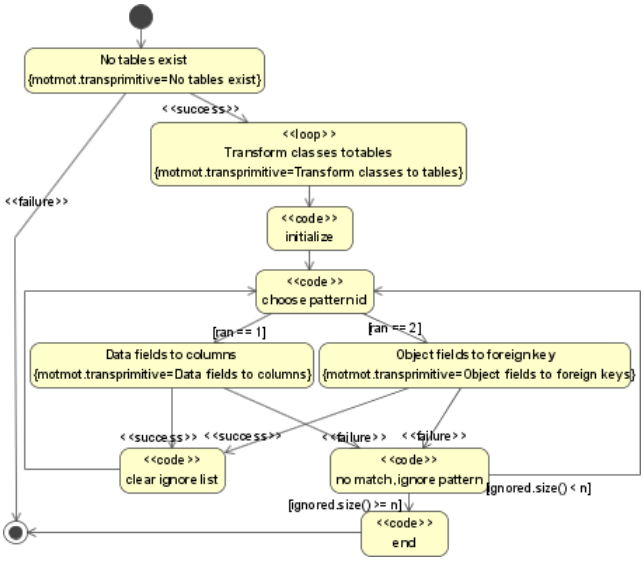


Figure 3: Fully imperative equivalent of Figure 2.

any order. Such rules need to be executed until all of them fail to match. Every time a rule of this set is evaluated, it is executed, and all the other rules of the set have to be checked again in the next iteration, because applying a rule to a model can change the model.

2.2 The fully imperative equivalent

The new construct must be transformed to an equivalent which is solely written in plain story diagrams. Otherwise, our contribution would probably only become supported by our own tool. Such an imperative equivalent for the example from Figure 2 is shown in Figure 3. After the `Transform classes to tables` state, the `initialize` state is entered, where some variables that will be used are declared. The ≪*code*≫ stereotype denotes that the state corresponds to (Java) code instead of a transformation rule. More in detail, an integer `n` is set to 2, as there are two rules (`Data fields to columns` and `Object fields to foreign keys`) that can be executed. A list `ignored` is initialized, which will represent the rules that must be ignored because they didn't match in the current model state.

After the `initialize` state, a rule is chosen non-deterministically in the `choose pattern id` state, also a code state. In fact, a random number generator produces an integer ranging from 1 to `n`, which represents the randomly chosen rule. Moreover, this generated integer must not be contained in the `ignored` list. According to this random number, one of the *transprimitive* states which represent the actual transformation rules is entered.

If the rule matches, the ≪*success*≫ transition is followed. This transition leads to a state that ensures that `ignored` is cleared, as the ignored rules must be evaluated again because the state of the model might have changed. Then, a new rule can be chosen in the `choose pattern id` state for execution.

If the rule didn't match, it would be useless that this rule would be coincidentally chosen in the next iteration, so the

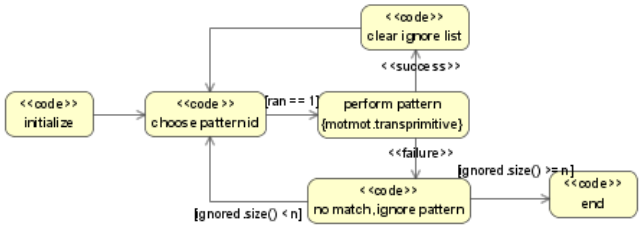


Figure 4: Prototype that is put into the input model.

rule is added to the `ignored` list. As long as there are rules available for execution, a new rule can be chosen for the next iteration. If there are no rules left for execution, i.e. they are all in `ignored`, the algorithm ends. The `end` state does nothing, but is added in order to end the equivalent with a transition without a guarding expression.

This results in an equivalent for unconstrained implicit rule scheduling, which schedules all the rules with equal priority. This algorithm can easily be extended to support constrained rule scheduling, like the use of priorities [2] (as realized in AToM^3) or layers [5] (as realized in AGG).

3. HIGHER ORDER TRANSFORMATION

In this section, a higher order transformation is proposed that transforms applications of the introduced new language construct to plain story diagrams.

The higher order transformation consists of a main loop that sequentially transforms each *transprimitiveND* state in the input model. Such states are transformed to their imperative equivalents as in Figure 3. These equivalents will be very similar to one another. Therefore, the common part is bundled into a separate generic prototype model, given in Figure 4. Starting from this prototype will avoid verbose rewrite rules in abstract syntax form. To use the prototype in the input model, it has to be moved to there. Roughly speaking, this is done in the following steps: first, the prototype model file is read. Secondly, all the states of the prototype model are moved to the input model. Thirdly, all the transitions of the prototype model are moved to the input model. Fourth, the target of some transitions is reassigned. And fifth, the source of some transitions is reassigned.

Interestingly, the second, third, fourth and fifth step can be executed independently. So in theory, a *transprimitiveND* state could be used to control the rewrite rules for these steps. Doing so would emphasize the similarity between these steps, thus improving the structure and readability of the overall transformation model. However, we have not yet *bootstrapped* the higher order transformation [4]. Instead, it is modeled using plain story diagrams (i.e., the language without the *transprimitiveND* construct).

Once a prototype is created in the input model, the (Java) code in the `initialize` code state must be changed in order to initialize `n` with the actual number of rules. Then, for each rule of the *transprimitiveND* state, a new `perform pattern` state has to be created, with its according transitions. Figure 5 represents a slightly simplified version of the pattern in the higher order transformation that performs this task. A new state `performState` and transitions from `choose pattern id` and to `clear ignore list` and `no match, ignore pattern` are created.

When each rule of the *transprimitiveND* state is added in a *transprimitive* state, a few elements, including the original *transprimitiveND* state must be removed from the input model. Once we bootstrap the higher order transformation, we will model this behavior using a *transprimitiveND* state with one rewrite rule for each element that needs to be removed. When all *transprimitiveND* states are transformed, the transformation completes.

4. RELATED WORK

The imperative realization from Section 2.2 executes rules sequentially, in a random order. Alternatively, one could execute the rules in parallel. It seems intuitive to rely on UML *Fork* and *Join* elements to model the parallel nature of a transformation system explicitly. However, neither version of Fujaba nor MoTMoT generates any code aimed at parallel execution (thread creation, synchronization, ...). Therefore, a higher order transformation approach such as the one presented in Section 3 does not seem to be applicable. Instead, one would have to extend the core of a particular story diagram tool. On the other hand, the use of standard UML elements (see Section 2.1) does apply. Instead of relying on UML activity diagrams as a standard language for controlling the application of rewrite rules, Syriani and Vangheluwe rely on DEVS [7].

The non-determinism of implicit rule scheduling can lead to unexpected results: in many cases, one rule for example creates elements that are used by another one and deleted by yet another rule. Since such dependencies can be introduced accidentally, dedicated analysis support is desirable in transformation tools that support languages with implicit rule scheduling. For example, the AGG tool offers a so-called Critical Pair Analysis (CPA [3]). To be applicable on the proposed hybrid language, CPA algorithms need to take into account nodes that are already bound from previously executed rules in a control flow.

5. FUTURE WORK

As an easy extension to this paper, the algorithm can be extended to support priorities or layers, as briefly stated in Section 2.2. Support for layers can be realized in another higher order transformation by putting together all rules of the same priority in *transprimiveND* states, then ordering them according to their priorities or layers. The higher order transformation from this paper can then transform the resulting model to a plain story diagram. In the case of priorities, the execution engine needs to re-evaluate all rules upon each iteration again, starting from rules with the lowest priority. Although this can be realized again using a higher order transformation, the one presented in this paper produces quite a different control structure.

The current realization aims at true nondeterminism by using random numbers. In some cases however, the order of the rules is irrelevant. More specifically, one may not care if the same order is used at all times. In these cases, the random number generator is nothing but a performance bottleneck. Therefore, we may extend our approach with another,

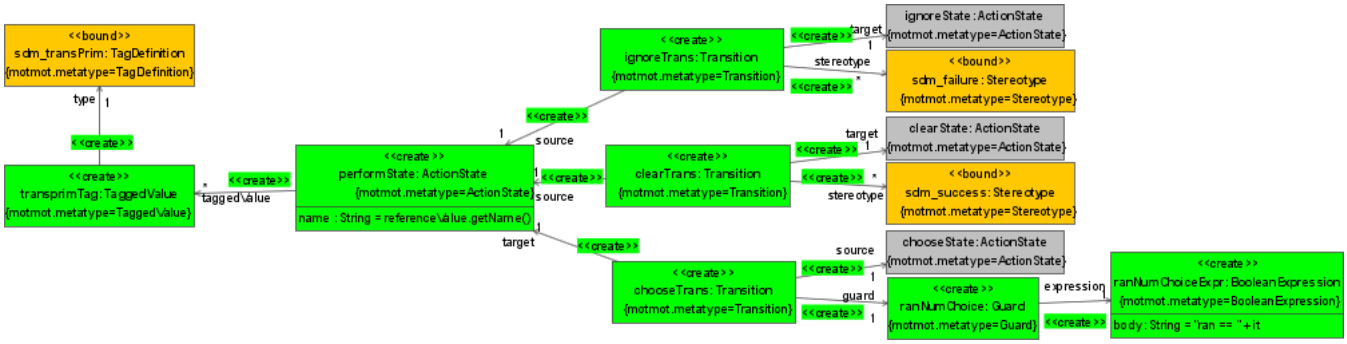


Figure 5: The rule in the higher order transformation that adds a perform pattern state and its transitions.

much simpler higher order transformation that imposes a particular order on the rules instead of guaranteeing randomness.

This paper discusses an algorithm for a *transprimitiveND* state in combination with a ≪*loop*≫ stereotype. Without a ≪*loop*≫, the *transprimitiveND* state ensures that all the rules are executed at most one time. A ≪*success*≫ transition will be followed if all rules did match once, and a ≪*failure*≫ transition will be followed if any of the rules did't match. As an example, suppose that in Figure 2, besides the `No tables exist` state, many other sanity checks have to be passed before the actual transformation can be done. All these checks could be referenced in one *transprimitiveND* state, avoiding a cascading effect of states with ≪*success*≫ and ≪*failure*≫ transitions, which could easily become very verbose and confusing.

This paper presents a transformation language that is hybrid with regards to rule scheduling. It is our ongoing work to realize a language that is hybrid with regards to other concerns, like *execution direction* and *change propagation* [8], too.

6. CONCLUSION

This paper discusses a standard syntax, the informal semantics and working tool support for a new language construct that allows users to use implicit rule scheduling in story diagrams. We illustrated its relevance and meaning by means of a toy example. As a more realistic example, we indicated where the language construct could even improve the readability and conciseness of its own compiler (i.e., that of its supportive higher order transformation). As a generic technique, this paper illustrated how profiles and higher order transformations enable language engineers to contribute new language contructs to a variety of tools (any version of Fujaba, MoTMoT, ...) without writing code specific to the editor or code generator of a particular tool.

7. REFERENCES

[1] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model transformations in practice workshop. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 120–127, 2005.

[2] S. M. Kaplan and S. K. Goering. Priority controlled incremental attribute evaluation in attributed graph grammars. In J. Díaz and F. Orejas, editors, *TAPSOFT, Vol.1*, volume 351 of *Lecture Notes in Computer Science*, pages 306–336. Springer, 1989.

[3] L. Lambers, H. Ehrig, and F. Orejas. Efficient conflict detection in graph transformation systems by essential critical pairs. *Electron. Notes Theor. Comput. Sci.*, 211:17–26, 2008.

[4] O. Lecarme, M. Pellissier, and M.-C. Thomas. Computer-aided production of language implementation systems: A review and classification. In *Software: Practice and Experience*, volume 12, pages 785–824, 1982.

[5] J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8:27–55, 1997.

[6] H. Schippers, P. Van Gorp, and D. Janssens. Leveraging UML profiles to generate plugins from visual model transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):5–16, 2004. Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation.

[7] E. Syriani and H. Vangheluwe. Programmed graph rewriting with DEVS. In M. Nagl and A. Schürr, editors, *International Conference on Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science, Kassel, 2007. Springer.

[8] P. Van Gorp. *Model-driven Development of Model Transformations*. PhD thesis, University of Antwerp, 2008.

Debugging Triple Graph Grammar-based Model Transformations

Mirko Seifert
Technische Universität Dresden
Software Technology Group
Dresden, Germany
mirko.seifert@inf.tu-dresden.de

Stefan Katscher
Technische Universität Dresden
Software Technology Group
Dresden, Germany
stefan.katscher@inf.tu-dresden.de

ABSTRACT

Model Driven Software Development heavily relies on expressive and formally founded model transformations. In this field, Triple Graph Grammars (TGGs) have not only been applied to practical problems, but shown that their formal grounding allows sophisticated analysis of transformation rules. However, creating a set of correct rules for a given transformation problem is not easy. Practical transformations inherently induce complex rule sets. Hence, developers of such model transformations need assistance to find faults in their rules.

This paper surveys existing debugging methods, derives a set of concepts that generally supports the localization and elimination of bugs and transfers these concepts to model transformations. In particular the peculiarities of Triple Graph Grammars will be addressed and essential ingredients for sophisticated debug support will be presented.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Design

Keywords

Triple Graph Grammars, Debugging

1. INTRODUCTION

Since the early days of computer science human errors became manifest in software of all kinds. The search for defects in programs has become a common activity of every software developer's life. Even though modern development tools significantly reduce the average number of bugs, producing zero defect software is still not practically achievable. Additionally, the ever growing complexity of today's software systems intensifies this problem.

Over the last decade, Model-Driven Software Development (MDSD) has been promoted as one promising way to tackle this complexity, targeting reduced effort and error rates in modern software systems. The idea is to use models to raise the level of abstraction in software design. The motivating assumption is that reducing the amount of information provided by developers is accompanied by lower defect rates.

MDSD replaces effort previously spent on coding programs in high level programming languages with other activities. Creating and transforming models, as well as generating code from them form the main activities within MDSD. Obviously, these new types of activities are not error-free by default. Due to the shift from code-centric development to model-driven processes, new kinds of defects arise. Erroneous source code is replaced by invalid, inconsistent or incomplete models, faulty model transformations or even buggy code generators. Consequently new methods and tools are needed to detect and remove defects.

If we focus on model transformations solely, we can spot a variety of existing approaches [2]. Both declarative and operational languages have been proposed and implemented. Among the former category, TGGs [7] provide a sophisticated way to specify and execute different model transformation tasks. Based on a formal grounding TGGs support bidirectional transformations, as well as model synchronization and integration.

However, having a well-founded and powerful formalism does not guarantee the correctness of concrete model transformations. Faulty transformation results can occur due to erroneous input models, a defect set of transformation rules or even a faulty implementation of the formalism. The contribution of this paper is to elaborate on existing approaches for debugging, derive a set of debugging principles or concepts and transfer them to the domain of model transformations. This will generate a set of requirements for model transformation debugging tools. Furthermore, TGGs will be examined with respect to their specific needs when debugging them. Thus, the paper tackles the detection of faults in transformation rule sets.

The remainder of this paper is organized as follows: Section 2 defines basic terms, recapitulates existing methods to debug imperative programs and derives the general concepts that are needed to support debugging activities. Section 3 shortly describes TGGs, in particular how they operate and highlights differences to other model transformation approaches that are important with respect to debugging. The main contribution of this paper can be found in Sect. 4, where we apply the concepts identified in Sect. 2 to model transformations and TGGs in particular. We conclude with a short outlook on future work in Sect. 5.

2. DEBUGGING CONCEPTS

Debugging is usually defined as the process of locating and fixing bugs (i.e., errors) in programs. To be more precise, errors are implementation mistakes made by humans which cause software to deviate from its expected behavior. Errors can cause a program to reach a state different from the correct one. If a program is in such a state, we say that there is a *fault*. Faults are not necessarily exposed to users because

programs may detect or compensate them either intentionally or by coincidence. If a fault is indeed propagated to the peripherals of a software (i.e., users or other systems) it is called *failure*. In this case, the expected result of a computation differs from the actual one.
Debugging was invented in the early sixties with the emerging of assembler languages. To localize erroneous behavior the concept of breakpoints was established and simulators were used to execute programs in a controlled environment. This allowed developers to stop the execution at particular instructions, which were considered to be relevant to the defect in a program. The developers inspected the current state of the program and tried to localize the instructions that were responsible for the failure. Later on hardware manufacturers picked up the idea and designed processors with built-in breakpoint support. This concept evolved over the last decades, but the general idea can still be found in almost every modern debugger.
The common goal of all debugging methods is to ease the localization of errors in programs. Thus, the power of a debugging technique is its ability to narrow the set of potential causes for a bug to the minimum. Traditional manual debugging with breakpoints is not very competitive in this sense, because developers must guess the locations of errors. Nonetheless, breakpoints are supported by almost all debugging tools for mainstream programming languages. However, these tools do not only provide support for controlling the execution of software, but allow to inspect and change the state of programs. Values of variables can be displayed and manipulated. Some tools even provide information about changes made to a program's state, for example by highlighting variables that have changed since the last stop of the execution.
To improve the assistance for manual debugging techniques researchers sought ways to allow developers faster access to potential causes for invalid program states. A technique called program slicing [8] was invented to determine parts of a program that are related to a given variable. The terms forward and backward slicing differentiate between slicing techniques to determine parts of a program that influence the value of a specific variable and parts that are influenced by such a value.
Inspecting a program's state is essential to find the point in time where a fault appears first. To relieve developers from determining this point by trial and error, a method called *Omniscient Debugging* [6] was introduced that records a program's state over time. Thus, a complete history for a single program run can be obtained and inspected.
More sophisticated approaches push the detection and localization of errors a little bit further. Static analysis tools save the effort of actually executing programs and localize potential and actual errors by solely looking at source or binary code. Based on symbolic execution [5] or abstract interpretation [1] tools can detect errors automatically. Other approaches define common bug patterns and try to detect these in suitable representations of programs [4].
From all previously mentioned methods and tools that support debugging we can derive a couple of general concepts (see Fig. 1). A program's state and its execution are the central objectives of debugging. In order to understand a program's behavior one must be able to control both. But, control by itself is not sufficient. To affect something in a meaningful way information about the program's logic

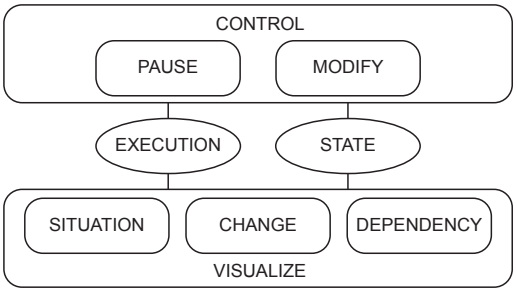


Figure 1: General concepts for debugging methods

and its current state is needed. Thus, visualizing the situation, changes to the situation, and dependencies is essential. Almost all approaches mentioned earlier, implement one or more of these concepts. Breakpoints allow to control a program's execution. Debuggers that allow manipulations to the memory of a program enable control of the program's state. Program slicing is used to determine dependencies between logic and data. Omniscient Debugging visualizes all situations a program encounters during a run.
The concepts shown in Fig. 1 abstract from specialized debug methods for textual or imperative languages. Rather, they provide general requirements for debugging tools. In Sect. 4 we will apply these concepts to model transformations and TGGs.

3. TRIPLE GRAPH GRAMMARS

TGGs were introduced by Andy Schürr [7] as an extension of Pair Grammars, providing a declarative way to express the relationship between two graphs. TGGs use three graphs, because in addition to the two graphs that are related (left and right graph), the third graph holds information about this relation. More precisely the third graph describes which nodes correspond to each other and is therefore called *correspondence graph*.
Based on the general notion of graph grammars, each TGG production has a left and a right-hand side. Both sides consist of three graphs, which can be connected. Since TGG rules are usually non-deleting (i.e., they only add nodes) it is common to visualize both rule sides in one graph.
The default form of TGG rules describes the relation between elements of both graphs (left and right) rather than what has to be done to transform one to another. Thus, this form is called the *relational form*. In order to perform a specific model transformation the rules must be converted to an *operational form*. Depending on the kind of transformation (left to right, right to left, synchronization or integration) the type of some nodes in the rule is changed. To transform from left to right, elements of type `create` of the left graph are changed to be `required`. Thus, the transformation engine searches matches for these elements in the left graph and creates elements in the right graph. Due to space limitations we must refer to [7] for more details.

4. DEBUGGING TRANSFORMATIONS

To apply the concepts from Sect. 2 to declaratively specified model transformations one must highlight some important differences to classic debugging. First of all, there is no predefined notion of an execution step. Imperative programming languages have an operational semantics that is

executed by some real or virtual machine. Usually instructions form the smallest unit of execution and consequently serve as the atomic execution steps. Even though, the underlying machine may perform multiple actions to execute one instruction, debugging abstracts from this detail.
To define a meaningful notion of execution step for model transformations, we must consider that transformation engines play the role of a virtual machine. The actual instructions executed by a transformation engine to find matches or to apply a rule are irrelevant to debug a transformation. A logical execution step must rather relate to some concept in the transformation language (e.g., nodes, links or constraints). We will present a set of such logical execution steps shortly in Sect. 4.1.
Another difference posed by some transformation languages is their graphical syntax. Textual syntax usually implies some order of reading, which often corresponds to the order of execution. Graphical rules have no such advantageous predefined order. Furthermore, finding meaningful visualizations for graphical structures is harder than for textual ones. This paper will not give a complete set of visualization methods that can be used in this context, but Sect. 4.2 and 4.3 will at least provide some starting points.
In addition to the specifics of general model transformations in the debugging context, TGGs have some properties which require special treatment. In contrast to imperative model transformation languages or basic graph transformation, TGG rules cannot be applied directly. An operational form of all rules must be derived from their original relational specification. As a consequence, debugging must incorporate this special preprocessing step.
Furthermore, the visualization cannot solely incorporate two independent models (source and target model), but must handle all three related models and the links between them. Existing editors for models can therefore not be applied as easy as in the general case.
A last property of TGGs that is important to debugging is non-determinism. Performing transformation based on TGGs in the most general form involves non-deterministic choices. Users trying to debug a transformation specification should have assistance to know about the choices available to the engine, see which choice is picked and be able to affect this non-deterministic decision if they want to.
This section gives more information about our notion of an execution step (Sect. 4.1) and explains how state (Sect. 4.2), change (Sect. 4.3) and dependencies (Sect. 4.4) should be visualized to support debugging activities.

4.1 Execution Steps

To obtain a natural notion of an execution step we use the basic elements of the TGG formalism (e.g., nodes, links, constraints, assignments). A step of execution must obviously involve at least one of them. For example, nodes must be matched, constraints are evaluated and assignments are performed. But, most elements are incorporated in different steps (e.g., nodes are matched, created and deleted). Thus, an execution step consists of a number of subjects (the involved elements) and a context (the action performed using the elements). We call this aggregation a *debug event*. Examples for events are: creating a node, matching a link in a rule to a link in a model, or changing the type of a node in a rule from `create` to `required`.
Figure 2 shows a simplified meta model for debug events.

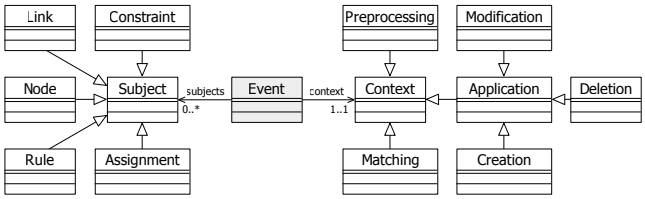


Figure 2: Meta model for debug events

This meta model is incomplete, because for practical use special subtypes (e.g., correspondence nodes) must be introduced. Nevertheless, the introduction of an artificial event concept is necessary to obtain meaningful units of execution. The implications of using such a scheme are manifold. First of all, transformation engines must explicitly create events and deliver them to debugging tools. This implies changes to existing engines, which do not have debugging facilities yet. In the best case these changes can be applied noninvasively (e.g., using AOP). For most existing engines some adaptation of the existing implementation will be needed. Our experience gained while extending our own prototypical implementation of a TGG engine showed that the effort needed is comparable with an extension of an existing application with detailed logging. Having said that, we must note that a common meta model for debugging TGGs can enable the independent creation of debug tools. These can in turn be used in conjunction with multiple TGG engines. As one can also tell from Fig. 2, we make heavy use of inheritance. This decision is driven by the need for extensibility. Introducing new special kinds of debug events should not invalidate existing debuggers. Rather, the event model should allow tools to handle general types of events and thereby process unknown events all the same.
Now that we defined some notion of an execution step, we can make use of it. Foremost, debug events can be used to define breakpoints and stop the execution of a transformation. Specifying constraints (e.g., on event types) allows for conditional breakpoints. This provides very fine grained control over the execution of a transformation. We can stop when model elements of a certain type are created, when the type of a specific rule node is changed, or basically when a complete match for a certain rule was found. Thus, we transferred one of the concepts mentioned in Sect. 2.
Furthermore, events can be collected and stored. Thus, omniscient debugging [6] can be applied to model transformations [3]. But, in contrast to recording state for imperative programs, an abstracted notion of the execution is used. This reduces the memory requirements, which is a serious problem for this debugging method. Filters that specify the debug events to be recorded can further relax this limitation. In any case, we obtain detailed information on the execution of a transformation. We have even used debug events to inspect a TGG implementation itself rather than a set of transformation rules. The events created while running test cases can be compared to expected values and indicate errors in the implementation itself.

4.2 State Visualization

Traditional debuggers support state visualization by providing users with a view on the memory used by a program. As TGGs operate on models and use rules, the state of a trans-

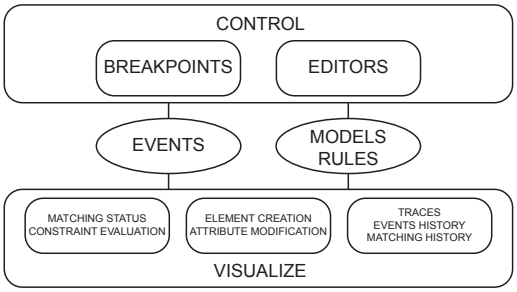


Figure 3: Elements for debugging TGGs

formation is the state of all involved models plus the rules. The involved models can be visualized using their existing graphical editors. However, visualization must incorporate the correspondence model. Coupling two existing editors (for the left and right model) with a third one (for the correspondence model) is probably a good option to do so.
Likewise, transformation rules must be available to users. But, rules must not be shown only, but adapted according to their current matching state. Assigning special color to matched elements is probably an intuitive way to provide this information to users. Changing the color of matched elements can also be applied to model elements.

4.3 Change Visualization

Visualizing changes that occur during a transformation is important to find defects in rule sets. Based on the events defined in Sect. 4.1, we can mark all model elements that were modified or added. Furthermore, accumulating a list of all modification events provides a history of all changes and the order they were applied in. This modification trace gives detailed information about the concrete effects of a rule set.

4.4 Dependency Visualization

Similar to the previously mentioned program slicing [8] dependencies can provide important clues about the location of defects. Knowing the set of possible causes for a certain effect can drastically reduce the effort needed for debugging. Speaking about model transformations, different kinds of dependencies are present. Newly created model elements depend on a set of elements in the source model. They also depend on the presence of a specific rule that matched these elements. If constraints are used they must be added to the set of dependencies as well.
To obtain all these dependency information we recommend to create traces for all new elements. These traces should contain details about the matched rule, the matched elements and checked constraints. Luckily, all this information is already present in the list of events created during a transformation run. Traces are basically an abstraction over the sequence of events that provide distilled information about dependencies. Using colors and existing editors for models and rules, can again support this visualization graphically.

Finally, a summary of all ideas mentioned above, can be found in Fig. 3. We replaced the concepts identified in Sect. 2 with their concrete counterparts for the context of model transformations.

5. CONCLUSION AND FUTURE WORK

After analyzing existing debugging methods and tools, we have derived a set of general concepts that are needed to support debugging activities. Controlling and visualizing the execution and the state of a process is important to find defects. We applied these concepts to model transformations by introducing debug events as a natural notion of execution step. We provided a simple meta model for debug events and explained how they support both control and visualization of a transformation's execution. Furthermore we collected a set of requirements on visualizing the state of a transformation (i.e., the involved models and rules). Some of the presented concepts can be applied to model transformations in general, others are specific to TGGs. Furthermore, we are convinced that agreement on a common debug event model can enable transformation independent debugger implementation.
In the near future we plan to finish the implementation of the mentioned concepts. This will allow us to gain practical experience and to validate the presented ideas. As the graphical syntax of TGG rules has some disadvantages (e.g., no predefined reading order), we want to create a textual counterpart. This implies other disadvantages (e.g., a syntactical gap between the models and rules that transform them). Thus, we will compare both syntaxes with respect to their usability for debugging activities.

6. ACKNOWLEDGMENTS

This work has been partially funded by the German Ministry for Education and Research in the SuReal project.

7. REFERENCES

[1] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[2] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture*, oct 2003.

[3] M. Hibberd, M. Lawley, and K. Raymond. Forensic Debugging of Model Transformations. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *LNCS*, pages 589–604. Springer, 2007.

[4] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[5] J. C. King. Symbolic Execution and Program Testing. *Communication of the ACM*, 19(7):385–394, 1976.

[6] B. Lewis. Debugging Backwards in Time. *The Computing Research Repository*, cs.SE/0310016, 2003.

[7] 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.

[8] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

Model Level Debugging with Fujaba

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

ABSTRACT

This paper presents an approach how story diagrams can be debugged on model level. Therefor it presents a technique how a model-code mapping can be created, how that mapping can be used with normal Java debuggers to enable model-based debugging and how integration with Fujaba4Eclipse works. Additionally it shows how this model-based debugging can be best combined with eDOBS for analyzing object structures in a debugging session.

1. INTRODUCTION

Fujaba enables the developer to specify his whole application on model level. For the structural aspects Fujaba offers class diagrams whereas for behaviour specification graph transformation in so-called story diagrams are used. From such models Fujaba automatically generates executable Java code. Normally the developer does not need to look at the source code since everything can be done in the model. If the developer discovers that his application does not show the wanted behavior or a test fails, he has to find the bug. This is normally done using debugging techniques. Since now the debugging needs to be done on source code level since Fujaba has no support for model-based debugging. Debugging on generated source code is especially difficult for developers who have no idea how the code generation works, e.g. students which are new to Fujaba. But also for experienced developers debugging on model-level would higher the abstraction level and therefor increases productivity and lowers the possibility of mistakes. This paper presents our solution and implementation for model-based debugging in Fujaba.

2. MODEL-CODE MAPPING

There exist two possible solutions to make model based debugging possible. The first one would be to interpret the model and visualize the results. Therefor one would need an interpreter for Fujaba's story diagrams. It would be hard to verify if the interpreter always has the same behavior as the generated Java code. Additionally one would not be able to use all the existing debugging tools that are e.g. integrated into eclipse. The second possibility would be to debug the generated code using a standard Java debugger and map the results back to model level. This approach is chosen in this paper.

For the chosen approach one first need a mapping which allows for a given line in the Java source code to find the model element which has generated this line of code. Therefor we integrate into the CodeGen2 code generator. CodeGen2 translates the model to an intermediate token layer, which is then optimized for code generation (e.g. a control flow analysis is performed). Afterwards for every token a template responsible for that token type is applied which finally results in the Java code written to file. See [1] for details on the code generation.

Each time CodeGen2 applies a template for a token, the generated code is now wrapped by a start and a end comment. This comments refers to an object of the type `DLRToken` using a unique identifier. That `DLRToken` stores references to the model elements that were associated with the token which generates that certain peace of code.

Listing 1 shows the source code of a simple `State` class. This code contains the generated comment for model-based debugging. One can see that the whole code is associated with a `DLRToken` with the unique identifier 6. This token represents the source code, that was generated for a file (an object of class `UMLFile`) in the Fujaba model. The code of the class was generated by a `DLRToken` with ID 7, the method `visit()` by ID 9, etc.

```
1  /*start id=6*//*
2   * generated by Fujaba - CodeGen2
3   */
4  package de.uni_kassel.statechart.model;
5
6
7  /*start id=7*/public class State
8  {
9  /*start id=8*/
10    private String name;
11
12    public void setName (String value)
13    {
14       this.name = value;
15    }
16
17    public String getName ()
18    {
19       return this.name;
20    }
21 /*end id=8*//*start id=9*/
22    public boolean visit ()
23    {
24 /*start id=10*/
25       /*start id=12*/return true;
26    /*end id=12*//*end id=10*/}
27 /*end id=9*/
28    public void removeYou()
29    {
```

```
30      }
31  }
32  /*end  id=7*/
33  /*end  id=6*/
```

Listing 1: Source code example with comments

Such source code may now be changed by some external tools like e.g. the eclipse code formatter or the EMF code generator with might generate additional constructs into the class. Afterwards the source code is reread and a tree of `DLRToken` is created according to the comments found in the file. Afterwards the file is written back without the comments. Figure 1 shows such a tree for the file of Listing 1.

The `d0` object of the singleton class `DLRTool` has a list with all open projects for which code was generated. Here this is just the project represented by the object `d1`. From there the tree is created by the `children` links. Every token in the tree knows the line of code where the code generated for that token starts and the line where it ends. Additionally it stores the character offset and the character length of the code block. Every token also has a `comment` attribute which stores a String describing what this code block does. The `comment` is not shown in Figure 1 to keep it readable. The token with ID 7 e.g. covers the lines from line 7 to line 32 in Listing 1. For that token the object of type `ElementReference` is also shown. This object connects the token with the corresponding model element. Here this is the `UMLClass` object `u16`. Using this tree every line in a Java file can be traced back to the model elements that originally created that line.

3. JSR-045 AND SMAP

The previous section described a code-model mapping. To now enable model-based debugging this mapping needs to be integrated into the debugging process. That means it should be possible to place breakpoints on model elements or to stepwise execute a story diagram. The Java VM offers an interface for such purposes. That interface is defined in JSR-045 *Debugging Support for Other Languages* [1].

JSR-045 is originally made to allow debugging a textual language other than Java using the standard Java debugger. To debug an application in a different language using JSR-045, for every source code file a so-called SMAP file must be provided. The SMAP file specifies which line in which file in the source language corresponds to which line and file in the target language which is normally Java.

To now debug Fujaba diagrams using JSR-045 one need to convert the mapping tree described in the previous chapter to a SMAP file. Note, that we are mapping a graphical language to a textual one here. But that is no big problem since our approach simply takes the IDs of the `DLRToken` in the tree as source language line number. Additionally, the tree has to be converted to a list. This is done by traversing the tree depth-first and relate every line of the target language to the token which generates this line and is deepest in the token tree.

[1] http://jcp.org/aboutJava/communityprocess/final/jsr045/index.html

The tree from Figure 1 would generate the SMAP file shown in Listing 2. The SMAP file starts with a header that states the target file (State.java, line 2) and the name of the source language (Fujaba, line 3). Line 6 holds the name of the target file and the mapping starts at line 8. A single mapping always looks like this: It starts with the start line in the source file (here the ID of the `DLRToken`). Then comes a "#" sign followed by the number of the source file (here always 1, see line 6). Then the number of repetitions might be given starting with a comma. This makes only sense for text-to-text mapping so it is not needed here. After a colon the target line number and separated by a comma the length of the code block is stated.

```
1  SMAP
2  State.java
3  Fujaba
4  *S Fujaba
5  *F
6  1 ..\..\..\..\StatechartDLR.ctr
7  *L
8  6#1:1,6
9  7#1:7,2
10 8#1:9,12
11 9#1:21,3
12 10#1:24
13 12#1:25
14 9#1:26
15 7#1:27,5
16 6#1:32
17 *E
```

Listing 2: SMAP file for the source code from Listing 1

The lines 9 and 15 in Listing 2 e.g. specify that the lines 7-8 and 27-32 in the Java file from Listing 1 are generated by the `DLRToken` with ID 7. That means that lines are generated by the `UMLClass` element in the Fujaba model.

The information from the SMAP file is now compiled into the Java class file using a post-compiler and can then be used by every debugger that supports the JSR-045, like the debugger integrated in eclipse.

4. FUJABA4ECLIPSE INTEGRATION

We have written a Fujaba4Eclipse plugin so that the approach described above can be used to debug Fujaba diagrams in Fujaba4Eclipse. Figure 2 shows that model-based debugger in action. Due to the SMAP integration it is now possible to add breakpoints to model elements, as shown in the Breakpoint View in Figure 2. The debugger just hit such a breakpoint. That is why the story diagram is opened in the Fujaba Editor and the corresponding model element is highlighted. The comment of the `DLRToken`, describing what would happen next, is shown in the status bar. In this case the debugger has stopped just before the `target` link is created. The Debug View in Figure 2 shows the stack trace of the suspended thread. Here it is possible to jump to elements higher in the stack trace or to change the debugger from model-based debugging to code-based debugging. In the Variables View the developer can analyze the local variables and the current object structure when the VM is suspended. It is somehow difficult to compare a pattern in a story diagram to the current object structure using the

Variables View to e.g. find out why a pattern fails. To solve this problem the eDOBS plugin described in the next chapter lifts the visualization of the object structure to model level.

5. EDOBS

eDOBS is an Eclipse plugin, which visualizes the current heap of a Java program at runtime as a UML object diagram [2]. eDOBS may typically be used within a debugging session. Instead of the variable view, the eDOBS panel may show the content of variables and how the referenced objects are related to each other. Figure 3 shows eDOBS visualizing the object structure while debugging the example from Figure 2.

To make the comparison of the current object structure in eDOBS and a story pattern in Fujaba easier, we have annotated all local variables names to the objects the variable currently stores. This is done by visualizing the names as notes behind the object name in eDOBS. In Figure 3 one could see, that the object `s0` is currently the `this` object, the object `s1` is stored as local variable `sc`, etc. Of course one object in eDOBS can have multiple variable names. Additionally, all objects currently not stored in any local variable, that means all object not visible in the current method block, are shown semi-transparent. This way it is quite easy to identify a pattern in the current object structure and e.g. analyze why the matching is not successful.

When exploring object structures in eDOBS, the way back to the Eclipse or Fujaba4Eclipse artifact is offered for all elements. That means that you can jump from an object / attribute / method in eDOBS directly to the source code or to the corresponding Fujaba4Eclipse diagram.

6. CONCLUSIONS

The paper has presented an approach to enable model-based debugging in Fujaba. The result is implemented as eclipse plugin in the CodeGen2 repository. It was tested with some small examples, e.g. the Fujaba Solution for the AntWorld Simulation Tool Case at the 4th International Workshop on Graph-Based Tools[2]. In our opinion model-based debugging helped alot when searching for bugs in those examples.

7. REFERENCES

[1] L. Geiger, C. Schneider, and C. Reckord. Template- and modelbased code generation for MDA-Tools. In H. Giese and A. Zündorf, editors, *Proc. of the third International Fujaba Days 2005, Paderborn, Germany*, volume tr-ri-05-259 of *Technical Report*, pages 57–62, Paderborn, Germany, September 2005. University of Paderborn.

[2] L. Geiger and A. Zündorf. eDOBS - Graphical Debugging for eclipse. *Electronic Communications of the EASST*, 1, 2006.

[2]http://www.fots.ua.ac.be/events/grabats2008/

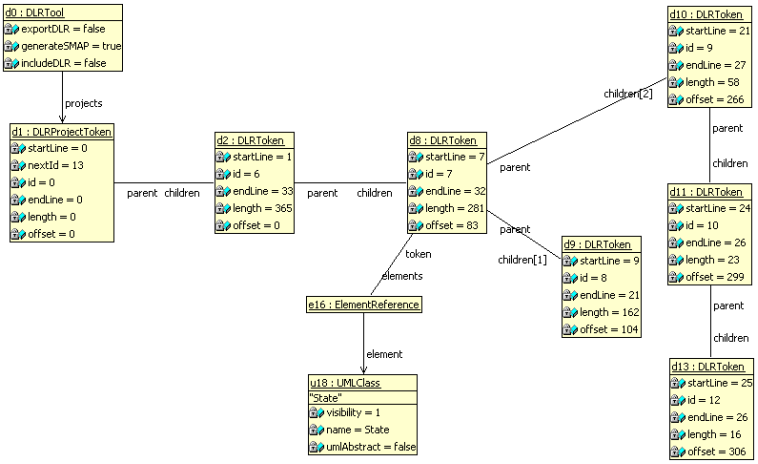


Figure 1: Mapping tree

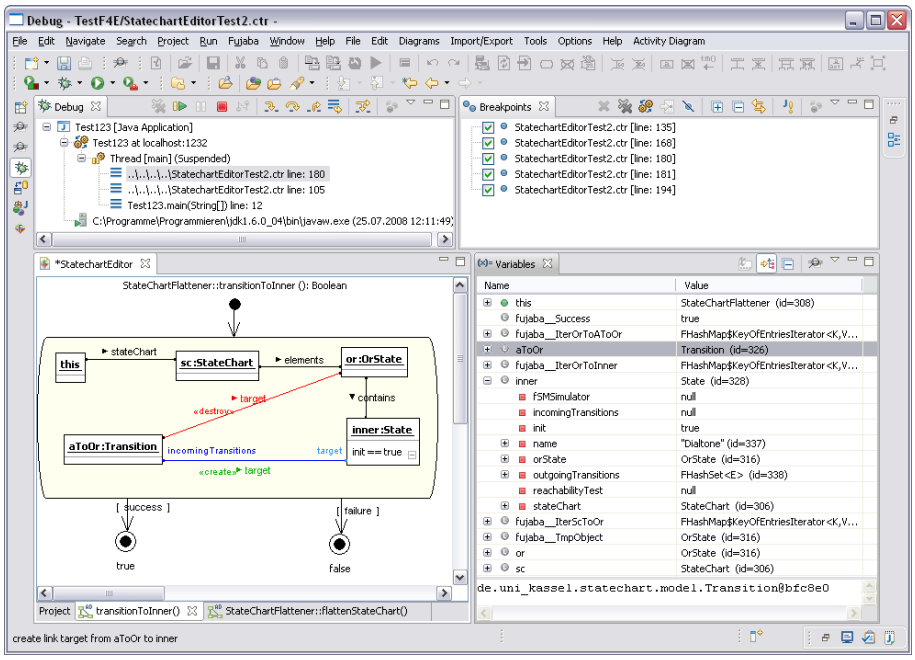


Figure 2: Model-based debugging in action

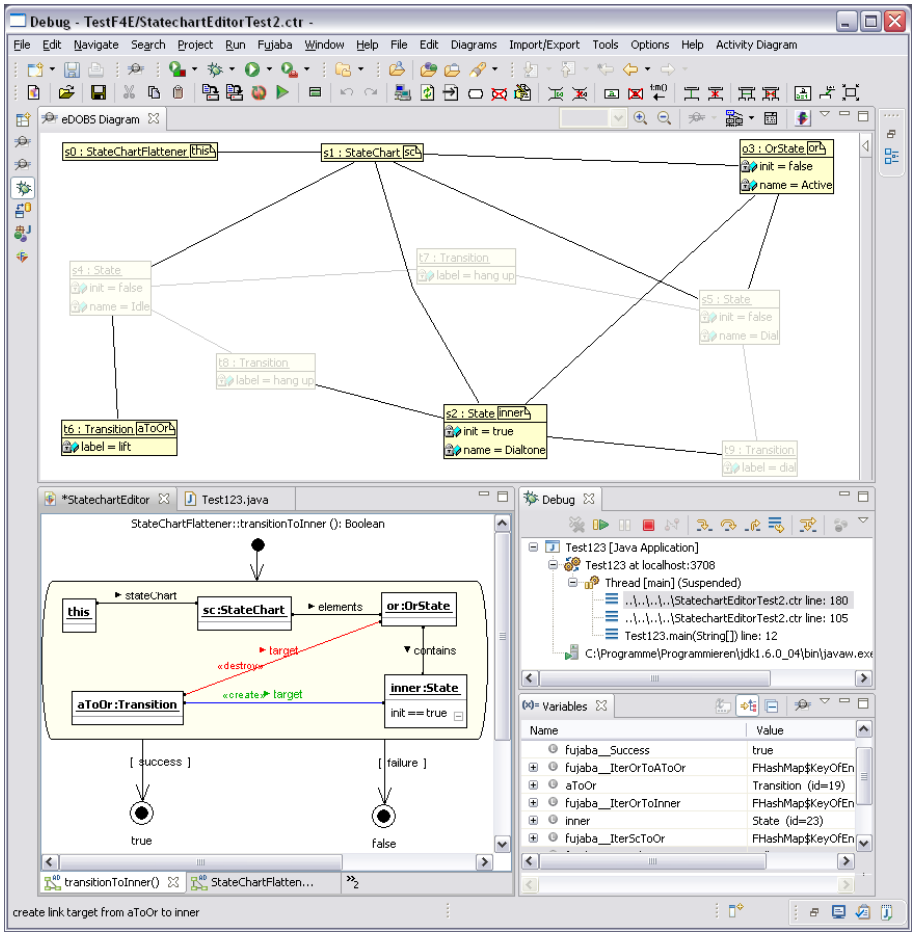


Figure 3: Model-based debugging with eDOBS

Fujaba's Future in the MDA Jungle

Fully Integrating Fujaba and the Eclipse Modeling Framework?

Basil Becker, Holger Giese, Stephan Hildebrandt, Andreas Seibel
Hasso Plattner Insitute for Software System Engineering
Prof.-Dr.-Helmertstr. 2-3
14482 Potsdam, Germany

{Basil.Becker|Holger.Giese|Stephan.Hildebrandt|Andreas.Seibel}@hpi.uni-potsdam.de

ABSTRACT

Fujaba has a long tradition as a CASE tool and provides outstanding features with its proprietary extensions in form of Story Diagrams and TGGs. However, in its current form Fujaba is not ready to support the development of tools or applications for MDA. In this paper, we report about our latest work on Fujaba and Eclipse and outline the used solutions developed and employed so far. In addition, we outline our vision of the next *open* Fujaba version, which more smoothly integrates with standard MDA environments and thus could help that the Fujaba concepts and our research results can be employed in standard development projects in the MDA world.

1. INTRODUCTION

Fujaba has a long tradition as a CASE tool and provides outstanding features with its proprietary extension in form of code generation for UML Class Diagrams [5], Story Diagrams [2] and TGGs [4]. However as outlined in this paper, Fujaba is currently not ready to support the development of MDA tools or applications. To employ Fujaba and its concepts, we have to restrict our projects to UML models encoded with the proprietary UML dialect supported by Fujaba. In practical applications, also other MDA techniques are required, which have to be used in combination. In fact, an *open* version of Fujaba is required, which can be employed in a well established technological platform.

To be compatible with such a platform, Fujaba must be able to support not only its proprietary UML models as basis for its concepts but also standard meta models and the standard APIs possibly generated by other non Fujaba code generators. The Fujaba offspring MOFLON [1] with its support for MOF 2.0 thus seems to be a good solution. However, neither MOF 1.x nor MOF 2.0 has found much acceptance in practice due to its complexity. Instead, the Eclipse Modeling Framework (EMF)[1] has become more or less the quasi-standard for meta modeling, which provides a partial implementation of the OMGs Essential Meta Object Facility (EMOF) in form of the Ecore meta meta model. In addition to basic support for modeling meta models and code generation from the Ecore meta models, EMF also provides model serialization/de-serialization and validation facilities. Furthermore, a large variety of other tools for the development of tools for MDA and DSLs based on EMF exist, e.g., openArchitectureWare[2], ATL[3] and GMF[4]. Also a number of modeling tools already employ EMF such as the UML2 Tools [5] for UML 2.0 modeling and TOPCASED[6] for SysML modeling.

In this paper, we report about our latest work on Fujaba and Eclipse and outline the used solutions developed and employed so far. In addition, we outline our vision of the next *open* Fujaba version, which more smoothly integrates with standard MDA environments and thus could help that the Fujaba concepts and our research results can be employed in standard development projects in the MDA world.

The paper is structured as follows: we first present our current achievements and requirements for the meta model based development of tools in Section 2 looking into meta modeling (Section 2.1), Story Diagrams (Section 2.2), Triple Graph Grammars (Section 2.3) and consistency rules (Section 2.4). Then, we discuss our requirements for UML 2.0 based development of applications in Section 3. Finally, we discuss our findings in Section 4 looking into open issues and our envisioned solution before our final conclusions.

2. META MODEL BASED TOOL DEVELOPMENT

2.1 Meta Modeling

Fujaba uses a proprietary meta meta model to represent meta models. This makes interoperation with modeling tools, based on other meta meta models, difficult. To create transformation rules for our model transformation system (described in Section 2.3), we first have to import the existing EMF meta models of the models, we want to transform, into Fujaba. Unfortunately, there was no possibility to import EMF based meta models into Fujaba, so far. Manually remodeling meta models is usually not an option due to the complexity of these meta models (e.g., UML 2.0). Therefore, we developed a simple import plugin for Fujaba, which uses EMF to de-serialize Ecore files and create the EMF model tree in-memory. This tree is then traversed in order to create a new Class Diagram, classes, attributes, methods, generalizations and associations into Fujaba.

[1] http://www.eclipse.org/modeling/emf/

[2] http://www.openarchitectureware.org/

[3] http://www.eclipse.org/m2m/atl/

[4] http://www.eclipse.org/modeling/gmf/

[5] http://www.eclipse.org/uml2

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

Problems arise if the Ecore model is split among several files. An example is the SysML meta model used in TOPCASED[7]. This meta model references the UML 2.0 meta model from the UML 2.0 plugin, which in turn references the Ecore meta model. Within Eclipse, platform URIs are often used to reference elements in models provided by other plugins. In Fujaba, these URIs cannot be resolved, of course. Therefore, the meta models have to be extracted from the providing plugins, put into the same directory, and all references in the files have to be made relative using a text editor.

Then, all models can be imported into Eclipse in the order in which they depend on each other. In the SysML case, first the Ecore meta model must be imported because it is independent from the other models. Then, the UML 2.0 meta model and finally the SysML meta model can be imported. The reason is, that elements referenced from another file are not converted to Fujaba by the importer. The Fujaba objects must already exist. After the import, the meta models are available for use in Fujaba.

2.2 Story Diagrams

Story Diagrams allow modeling behavior, using graph rewriting rules. They are similar to Activity Diagrams extended by special Story Activities that contain graph rewriting rules. In Fujaba, Story Diagrams specify the behavior of a method related to a class. A code generator can generate Java code from these Story Diagrams that implements the modeled behavior.

Nevertheless, a problem occurs with meta models imported from EMF. In Fujaba, compositions are always bidirectional. This is not required in EMF. EMF provides the method **eContainer()** to access the container of a contained element. Importing a unidirectional EMF composition into Fujaba, automatically creates an explicit inverse association from the contained element to the container. Of course, this association does not exist in the EMF model or the code generated by EMF. This poses a problem for Fujaba's code generation from Story Diagrams because the code generator might use these *non-existent* associations in the generated code. This makes the generated code incorrect and it must be corrected manually.

2.3 Triple Graph Grammars

In the context of Model-Driven Development (MDD), various different models describe a system under construction. These models can be used to describe different aspects, subsystems or overlapping parts of the system at different levels of abstraction. Thus, models are not completely independent but rather relationships between them exist, e.g., models are derived from other models by means of model transformations or they are kept in sync by means of model synchronization. Model transformation and synchronization require a transformation system to transform different model types and propagate changes between models. A model transformation system must be seamlessly capable of being integrated into an existing tool chain.

The Triple Graph Grammar (TGG) based model transformation system introduced in [3, 4] has recently been migrated to Eclipse. Thus, TGG based model transformations

[7]We encountered this problem in an industrial development project where a model transformation for SysML models was developed.

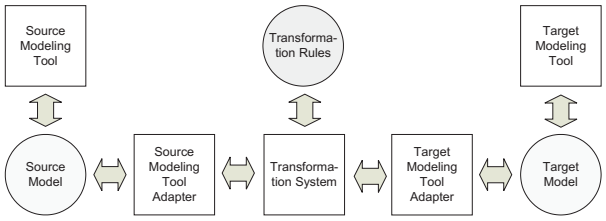


Figure 1: Overview of an in-memory transformation

can be easily used within Eclipse. This integration also allows exploiting EMFs change notification mechanism for efficient synchronization of models. However, the transformation rules, that perform the actual transformation, are not created within Eclipse but within Fujaba by means of Story Diagrams.

2.3.1 Model Transformation on EMF Models using Triple Graph Grammars

The model transformation system consists of a series of Eclipse plugins. Core of the transformation system is the transformation engine. It loads the source model of the transformation and the required set of transformation rules to create a target model. The engine is independent from the model types. It just executes the transformation rules, which perform the actual model transformation. The transformation rules are therefore specific for two types of models, the source and the target model. Transformation rule sets are deployed as separate Eclipse plugins.

The TGG-based model transformation algorithm supports bidirectional model transformation and synchronization. The transformation can be executed on two conceptual levels: file-based transformations and in-memory transformations. The file-based transformation loads the source model file from disk and saves the transformed target model to disk, as well.

The in-memory model transformation (shown in Figure 1) directly manipulates the models in the modeling tools' memory. For this purpose, a special tool adapter is required that allows accessing the models. If the modeling tool is another EMF-based Eclipse plugin, such an adapter is usually quite simple. In that case, the adapter just has to open the modeling tool, get the resource object that contains the model, and return it to the transformation engine. The transformation engine can then operate directly on the model. If the modeling tool is an external program, the adapter is much more complex and must translate modifications from the EMF modeling space to the external modeling tool, e.g., via COM or .NET whenever the external tool provides an appropriate interface. To execute a model transformation, the user has to provide a source and target model and a set of transformation rules.

2.3.2 Creating Model Transformation Rules with Fujaba

Figure 2 shows the main steps, which are required to create transformation rules within Fujaba. Before the actual transformation rules can be defined, the meta models of the source and target models must be available in Fujaba. In case of Ecore meta models, these must be either remodeled or imported into Fujaba. Besides the source and target meta models, a third correspondence model is required for a model transformation with TGGs. This correspon-

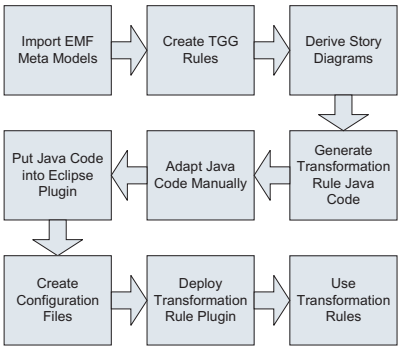


Figure 2: Necessary steps to create executable transformation rules from TGG rules

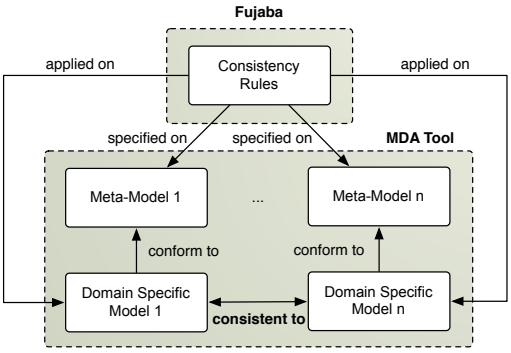


Figure 3: Conceptual integration of Fujaba and an Eclipse-based MDA tool for consistency rule specification

dence model captures traceability links between corresponding source and target model elements. Of course, a correspondence meta model is required that describes the available types of correspondence nodes and their inheritance relationships. In addition to the meta models of the models to be transformed, the meta model of the transformation system itself is required. This meta model is also available as an Ecore model and must be imported into Fujaba, as well.

When all necessary meta models are available in Fujaba, the TGG rules can be specified with the Fujaba TGG editor. While these TGG rules are only declarative, operational transformation rules have to be derived afterwards in form of Story Diagrams. For each TGG rule and each transformation direction a separate Story Diagram is derived. The generated Story Diagrams are the basis for the subsequent code generation. The code generator has to generate proper EMF compatible access methods for object creation and modification. For example, factories create all EMF objects. These factories must be used in the generated code to create new EMF objects.

The current code generation (CodeGen2 plugin) of Fujaba already supports the generation of EMF compatible code. To enable the EMF code generation, the code style attribute of all model elements must be set to "emf". The generated Java code is then integrated into an Eclipse plugin project. Due to the issue described in Section 2.2, several manual corrections of the code may be necessary. In the mentioned industry project, this issue posed a huge obstacle because the UML 2.0 meta model contains mostly unidirectional compositions and the code generator extensively uses the false inverse associations. These errors had to be corrected manually in the generated code.

Besides the classes for the actual transformation rules, some more helper classes and configuration files need to be created manually that, e.g. describe the type of models the transformation rules can be used for. Especially, the transformation rule plugin has to implement an extension point provided by the transformation system. The transformation system employs this extension point to discover available transformation rules. Currently, these configuration files have to be created by the user but rather should be created automatically.

2.4 Consistency Rules

We are currently working on another MDA application, which is a tool for model-based development of UML based applications deployment where we employ the notion of component development and component configuration & deployment. The developed software components provide software development aspects as well as deployment aspects like well defined variation points for configuration during deployment.

Fujaba does not provide the necessary modeling artifacts for this problem domain. In contrast, Eclipse does provide with EMF an appropriate environment for defining meta models, based on Ecore. Thus, we employ Eclipse and EMF as modeling environment. However, Fujaba is still a major part of the tool because we define consistency rules between domain specific models by means of Story Diagrams. Figure 3 shows the tool, its artifacts, the role of Fujaba and the relationships between these artifacts.
Consistency rules have to be specified on the meta models and are subsequently applied on the domain specific models, which are conform to their meta-models (instantiations). If relationships between domain specific models exist, which is quite common in our case, consistency rules are applied in order to check whether all required domain specific models are consistently modeled.

In order to realize such a tool, we employ the previously explained EMF importer and the EMF code generation capability of Fujaba's code generation. Thus, we import the EMF meta models into Fujaba, specify the consistency rules based on these meta models and subsequently generate EMF compatible code for the consistency rule application within Eclipse.

2.5 Open Issues

The major issue is the different meta meta models of Fujaba and EMF. This requires a conversion of meta models of one type to another. However, this conversion does not work seamlessly because of the major conceptual differences. One mentioned problem is compositions, which are always bidirectional in Fujaba. This makes the generation of *correct* Java code in Fujaba difficult.

Furthermore, the described workflow for creating a set of transformation rules is far too complex for an ordinary user of the transformation system. The process involves two different platforms, Eclipse and Fujaba, and requires a lot of manual steps. This workflow must be limited to only one platform and be automated as much as possible.

3. UML 2.0 BASED APPLICATION DEVELOPMENT

Fujaba does not allow UML 2.0 based application development, yet. It is missing the capability of modeling component based systems, a feature, which is nowadays of major importance. Of course, it is possible to add components and all other missing modeling aspects to Fujaba's meta meta model, but this is related to huge implementation efforts. Obviously, this reduces or even circumvents Fujaba's usage in industry cooperation's, as industry mostly uses and arrogates UML 2.0 conform models, respectively.

Story Diagrams can only be used with Fujaba models and within these models only for objects. But using them for other modeling artifacts providing a similar instantiation concept, i.e. components, would be a great benefit. However, it is impossible to use the modeling capabilities of different MDA / CASE tools and then use Fujaba to extend these capability by means of Story Diagrams. Following, if a developer wants to use Story Diagrams, she/he is currently locked in Fujaba.

4. DISCUSSION

The sketched workflow of Section 2.3.1 is not quite optimal. Two different platforms and meta meta models, Fujaba and EMF, have to be used. Fujaba does only support Class Diagrams of UML 2.0, which are even not completely compatible to UML 2.0. The import of EMF models into Fujaba and the code generation of Fujaba do not work perfectly, yet. Necessary configuration files cannot be created automatically at all and several manual steps are required. Furthermore, TGGs and Story Diagrams are not available as EMF models, which would allow model transformations on these models using the explained transformation system. The main cause for these issues is the gap between the meta meta models that must be bridged somehow.

To overcome these problems, the tool chain should be limited to a single platform. Because EMF is already widely in use, it is the canonical choice. However, this requires the migration or redevelopment of editors for TGGs and Story Diagrams as well as a code generation from Story Diagrams. Although Fujaba4Eclipse already tried to integrate Fujaba within Eclipse, the migration did only change the user interface but the underlying technology is still incompatible to EMF.

Therefore, reusing established technologies of the Eclipse world must leverage the integration of Fujaba and Eclipse. Reusing EMFs widely used meta meta model would make Fujaba compatible to other EMF based tools and would also reduce maintenance effort. Such tools could also be reused and do not need to be redeveloped for Fujaba anymore.

This also opens another possibility for the generation of Story Diagrams from TGG rules as explained in Section 2.3.2. Up to now, the generation of Story Diagrams from TGG rules is hard coded. This complicates the subsequent development of the transformation algorithm because a large part of the algorithm is contained in the operational transformation rules. If the transformation algorithm is modified, the generation algorithm for the Story Diagrams has to be changed, as well. If TGG and Story Diagrams were also EMF models, the transformation algorithm itself could be used to create the Story Diagrams. Only a set of transformation rules has to be provided that transform TGGs into Story Diagrams. To modify the transformation algorithm, only this set of transformation rules has to be adapted.

To achieve the described goal, we are currently working on the development of a Story Diagram editor based on GMF. Because Story Diagrams are quite similar to UML Activity Diagrams, we simply extend the UML 2.0 meta model provided by the UML 2.0 plugin by reusing intersecting parts. Subsequently, a code generator for Story Diagrams must be developed because a Story Diagram's semantic is expressed in terms of the resulting code. EMFs code generation mechanism can be reused to generate the main structure of the code but a tailored code generation is required to generate code from the behavior modeled by means of Story Diagrams.

A drawback of the sketched solution is that story-driven modeling is then limited to Ecore models. Although, Story Diagrams and their associated Class Diagrams are based on UML 2.0, these Class Diagrams have to be converted to Ecore models in order to use EMFs code generation. This is a restriction because Ecore provides only a subset of the constructs supported by MOF or Fujaba. A code generator that can generate Java code directly from the UML 2.0 model without a prior conversion to Ecore can solve this problem.

5. CONCLUSION

In this paper, we have presented two major issues that appeared in our recent work. First, the future of meta-modeling seems to be hold by EMF and second, Fujaba's Story Diagram and TGG features are important for engineering quality software. We have experienced and explained that Fujaba provides only poor compatibility to EMF and therefore is going to loose track of further developments in MDD. To overcome this, we propose to open Fujaba towards EMF and benefit from EMFs success on the one hand and enabling Fujaba's key technologies on the other hand, such as Story Diagrams and TGGs for EMF based development.

6. REFERENCES

[1] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *LNCS*, pages 361–375. Springer Verlag, 2006.

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

[3] H. Giese and S. Hildebrandt. Incremental model synchronization for multiple updates. In *Proceedings of GraMoT'08, May 12, 2008, Leipzig, Germany*, volume Proceedings of GraMoT'08, May 12, 2008, Leipzig, Germany, 2008.

[4] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. In *Software and Systems Modeling*, 28 March 2008.

[5] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 742–745, New York, NY, USA, 2000. ACM Press.

Letting EMF Tools Talk to Fujaba through Adapters

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

ABSTRACT

Many software modelling tools are built on top of the Eclipse Modeling Framework (EMF) through which they can communicate and exchange models. In contrast to that, the Fujaba Toolsuite defines its own modelling framework. Both frameworks are built on the same concepts of software modelling. Therefore, they can be adapted. This paper presents an implementation of a generic adapter layer that adapts Fujaba's modelling framework to EMF. Through this adapter layer, Fujaba models can be processed by any EMF-based tool without adapting each of those tools individually.

1. INTRODUCTION

The Fujaba Toolsuite and the Eclipse Modeling Framework (EMF) are both extensible software modelling frameworks. Fujaba, as an academic tool, has a long history and grew since the beginning of UML modelling into a set of tools based on a common core framework. EMF, as an industrial driven open-source framework, has attracted a larger community and consequently many modelling tools were built based on it during the last years. Today, Fujaba and EMF are both stable and productively usable modelling technologies, where each has its advantages and disadvantages for specific software modelling tasks.

In Model-Driven Software Development (MDSD), which attempts to use modelling during the whole software development process, tool integration is very important. Focusing on Fujaba and EMF-based tools, both could be used together and profit from each other in MDSD processes. Therefore, solutions are needed to integrate them tightly.

In earlier works, several ideas were presented that deal with integration and exchange between Fujaba and EMF. They were either based on aligning Fujaba-code itself [8] or on model-transformations that convert Fujaba to and from EMF models [5, 7]. The earlier require invasive changes of the Fujaba source-code, which is problematical when the tool evolves; the latter have a static nature, which requires explicit translations that hinder a smooth runtime integration of both tools and easily lead to data inconsistencies. The approaches so far also did either not succeed or not attempt to provide a generic integration of both frameworks.

In this paper we apply the well-known Adapter Pattern [4] to implement a small set of adapters that mediate between Fujaba and EMF tools at runtime. Therefore, no changes neither of Fujaba nor of EMF are required, but models are kept synchronised at runtime. The only premise for such adapters is that both tools expose the runtime model instances to the outside—which both do.

The paper is structured as follows: Section 2 shows the points at which Fujaba and EMF need to be adapted and describes our adapter implementation. The usability of the implementation is demonstrated using different examples that show how Fujaba interacts with different EMF-based tools in Section 3. Section 4 concludes and discusses possible enhancements of the adapter layer.

2. MAPPING AND ADAPTATION

In this paper, we focus on inspecting and modifying Fujaba models with EMF tools, which can only handle EMF models. To let Fujaba models look like EMF models, the following four concepts, found in both Fujaba's and EMF's implementation of *model*, need to be adapted: A model in Fujaba or EMF (1) conforms to a metamodel, (2) consists of model elements and (3) references between the elements, (4) is instantiated by a factory, and (5) persisted in a resource (e.g., a file). This section shows how the different implementations of the five concepts can be adapted.

2.1 Metamodel Mapping

A metamodel defines the concepts of a modelling language and can therefore be used to instantiate a modelling framework for that language. Since we want to access Fujaba models from EMF, the first step is to make Fujaba's modelling language—UML Class Diagrams, Statecharts, and Activity Diagrams with Story Patterns—known to EMF. This can be done by providing a metamodel of these languages in Ecore[1] format. The second step is to map the metaclasses of this metamodel (i.e., Fujaba metamodel in Ecore format) to the corresponding representations of the metaclasses in Fujaba (i.e., Java classes implementing the metaclasses).

To solve this issue, we implemented a tool that extracts information about the Fujaba metamodel from the Fujaba class files using Java's refelection facilities. For each Java class that belongs to Fujaba's metamodel implementation, the tool constructs an `EClass` (EMF metaclass representation) and organises all of them in `EPackages` (EMF metamodel representations), which are then registered in EMF's metamodel registry. For this, the tool makes assumptions about Fujaba's metamodel implementation but also requires additional information that can not be derived from the implementation allone. Describing all the assumptions and additional parameters in detail is not possible in this paper due to space limitations. Some assumptions are, however, presented in the context of Sections 2.2 and 2.3.

[1]EMOF [9] conformant metamodelling language of EMF

During the extraction, a mapping between the Fujaba metaclass representation (Java class objects) and Ecore metaclass representations (`EClass` objects) is created. The mapping is used by the adapters described in the next sections.

2.2 Model Element Adapter

The most important objects to adapt are the ones representing model elements. In both Fujaba and EMF, each model element is represented by one Java object. In Fujaba, all those objects are instances of a class implementing `FElement`. In EMF, the objects are instances of a class implementing `EObject`. Both interfaces offer convenient functionality for each model element; for instance comparability or listener support. In Fujaba, the Java class of which a model element is an instance corresponds to that element's metaclass. In EMF, this does not have to be the case.[2] The metaclass of an element can always be determined by the `eClass()` method of `EObject`. A standard implementation of `EObject` that can be used to represent any model element is `DynamicEObjectImpl`.

This loose coupling between metaclasses and their Java implementation is possible in EMF because the `EObject` interface offers rich reflection capabilities to inspect and modify models. For example, `eGet(String feature)` delivers the value of a feature by naming it; `eSet(String feature, Object value)` sets a feature to the given value. In Fujaba, such reflection capabilities are not provided by the `FElement` interface. Models can only be inspected and modified by using Java methods that correspond to element features (e.g., `get<featureName>()`).

Our adapter implementation delegates the reflective methods of `EObject` to methods of Fujaba metamodel classes using Java reflection. It hereby makes assumptions about the method names in Fujaba metamodel classes which were already used in the metamodel mapping (Section 2.1) and correspond to method naming applied by the Fujaba code generator.

The dynamic model element adapter, `DynamicEObject4FujabaModels`, is implemented as an extension of the existing `DynamicEObjectImpl`. The `DynamicEObject4FujabaModels` constructor expects an instance of `FElement`. This is the adapted Fujaba object to which the constructed adapter is bound for its lifetime. Furthermore, the methods `dynamicGet()` and `dynamicSet()` are overridden that handle reading and writing properties of the object. They realize the connection between the public `EObject` interface and the storage of information.

`dynamicGet(int featureID)` is implemented as follows:

1. Use the featureID (an identifier for a `EStructuralFeature` object that represents a feature defined by an `EClass`) to obtain name, mutiplicity, and type of the feature
2. If the feature's multiplicity is 1: find and call the method `get<featureName>()` on the adapted fujaba element (call the method `is<featureName>()` instead, if the type is `boolean`)
   (a) If the called method returns a String, Integer, or Boolean value, return that value
   (b) If the called method returns an `FElement`, ask the `DynamicEObject4FujabaModelsFactory` (cf. Section 2.4) for the corresponding Adapter and return it
3. If the feature's multiplicity is > 1: if no `DynamicEList4FujabaModels` representing the multiplicity feature was constructed yet, construct one (cf. Section 2.3); return the `DynamicEList4FujabaModels` representing the multiplicity feature

`dynamicSet(int featureID, Object value)` is implemented as follows:

1. Use the featureID to obtain name, mutiplicity, and type of the feature
2. If the feature's multiplicity is 1:
   (a) If the type is an `EClass`, ask the metamodel mapping (cf. Section 2.1) for the corresponding Fujaba metamodel class and use that as type (otherwise the type is String, Integer, or Boolean and can be used as is)
   (b) Find and call `set<featureName>(<featueType>)` on the adapted fujaba element
3. If the feature's multiplicity is > 1: do nothing (handled by `DynamicEList4FujabaModels`, cf. Section 2.3)

2.3 Collection Reference Adapter

Features with a multiplicity > 1 have to be handled explicitly in modelling tools implemented in Java, since there is no notion of attributes with a multiplicity > 1 in Java directly. Instead, collection objects have to be used. Fujaba and EMF handle this differently: Fujaba offers three[3] methods for each multiplicity > 1 feature directly on the corresponding metaclass. These methods are `iteratorOf<featureName>()` (returns an iterator over the feature), `addTo<featureName>(value:<featureType>)` (adds an element to the feature), and `removeFrom<featureName>(value:<featureType>)` (removes an element from the feature). In EMF, no such methods are defined by metaclasses. Instead, a list object is returned when the value of a specific feature is requested. Clients can add/remove elements to/from the list which directly manipulates the feature. Therefore, EMF comes with its own extension of the Java Collections Framework to insert additional functionality into the list methods which are required for model manipulation. In particular, lists representing features have to implement the interfaces `EList`.

For our adaptation, we require an `EList` that delegates all operations to methods of the adapted Fujaba object. In the implementation `DynamicEList4FujabaModels`,it was not possible to reuse an existing implementation (as for the model element adapter), since the functionality of storing data is so fundamental to Java's list implementations that nearly every methods accesses the data storage directly. Consequently, we implemented the required interfaces directly and did the following delegations to the adapted Fujaba element:

[2]It can be the case if EMF's code generation is applied

[3]There are more methods to access features, but the enumerated three are sufficient. Availability of other methods also varies between metaclasses.

1. `iterator()` delegates to `iteratorOf<featureName>()`
2. `add()`delegates to `addTo<featureName>()`
3. `remove()` delegates to `removeFrom<featureName>()`

The adapted Fujaba element—and the feature that is adapted by a `DynamicEList4FujabaModels`—are both passed to that list in its constructor, when it is created by a `dynamicGet()` method call of a `DynamicEObject4FujabaModels` (cf. Section 2.2). All other list operations dictated by the implemented interfaces are based on the three that delegate to the Fujaba element.

2.4 Model Element Factory Adapter

The two presented adapters are used to access and manipulate existing model elements. What is not supported yet is the creation of new elements, which is implemented in the `DynamicEObject4FujabaModelsFactory`. Both Fujaba and EMF use factories for model element creation and have a registry for factories that can be queried for a suitable factory for a given metaclass. Therefore, adaptation is straight forward: `DynamicEObject4FujabaModelsFactory` extends EMF's standard factory implementation `EFactoryImpl` by overriding the `basicCreate(EClass)` template method. Our implementation uses the metamodel mapping (cf. Section 2.1) to obtain the Fujaba metaclass corresponding to the given `EClass`. It then asks Fujaba for a factory suitable for that metaclass and uses that factory to create a new Fujaba model element.
A singleton `DynamicEObject4FujabaModelsFactory` is registered at the metamodel created by the mapping (cf. Section 2.1). Doing so forces EMF to use this factory for creating elements conforming to the corresponding metamodel.
The `DynamicEObject4FujabaModelsFactory` also acts as a model element adapter (cf. Section 2.2) registry. It is used by all four adapter types (cf. Sections 2.2, 2.3, 2.4, 2.5) to obtain an adapter for a Fujaba model element. If the adapter does not exist yet, it is created for the corresponding element and registered.

2.5 Resource Adapter

The last missing piece in the adaptation is the storing and loading of models. Fujaba models are loaded and stored by Fujaba and made available in Fujaba's workspace. In EMF, so called `Resources` are used to represent physical storage. We leave the loading and storing of models to Fujaba and adapt it as well, by providing a `FujabaResource` that, instead of loading from and writing to a physical storage, accesses the Fujaba workspace.
Resource types can be registered at EMF for a file extension. We register the `FujabaResource` for the extension `.fujaba`. When a file with the `.fujaba` extension is opened in the Eclipse workspace, EMF attempts to load it as a `FujabaResource`, which takes the file name of the opened file and looks for a Fujaba project (in Fujaba's workspace) with the same name. It then retrieves the adapter for that project (which is also a Fujaba model element) from the registry (cf. Section 2.4) and sets it as the resource's content. Therefore, any Fujaba project can now be accessed as EMF model from Eclipse by creating an empty file somewhere in the Eclipse workspace using the scheme: `<FujabaProjectName>.fujaba`.[4] Saving works in a similar fashion by manipulating the Fujaba workspace in `FujabaResource`'s saving method.

3. APPLICATIONS

This section shows how the adapter is used by different EMF-based tools without further effort. Its aim is to demonstrate the rich possibilities of a generic integration of Fujaba and EMF as demonstrated in the last section. Throughout this section we use s simple UML model modelled in Fujaba shown in Figure 1.

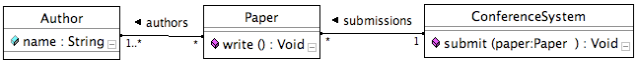


Figure 1: Model of a conference system in Fujaba

3.1 Displaying Fujaba Models in EMF-style

The first simple application is a tree model editor included in EMF. This editor can display any model independent of its metamodel. It uses the containment relationships between model elements to determine the tree structure. Figure 2 shows the example opened in that editor. Note that we can not only inspect, but also modify the model with the editor.

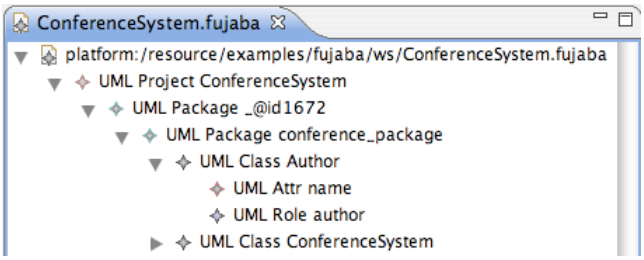


Figure 2: Fujaba model displayed in EMF's editor

3.2 EMF Compare: Diffing Fujaba Models

EMF Compare [3] is a tool that computes diffs between two versions of a model and visualises them using tree representations as used by the editor above. Imagine that we import a new version of the example into the Fujaba workspace under the name ConferenceSystemNew, where the class Paper has been renamed to Submission. EMFCompare suggests that Submission was indeed Paper before by inspecting both versions' structures as shown in Figure 3.

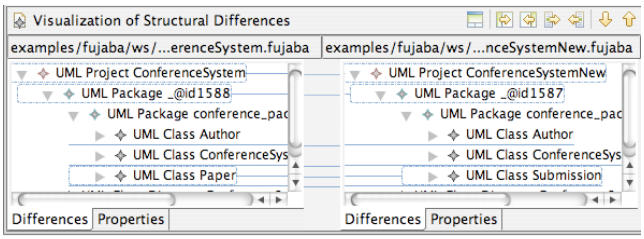


Figure 3: Visual diff between Fujaba models

[4]Fujaba has to run in the same JavaVM as Eclipse such that the Fujaba workspace can be accessed.

3.3 Transforming Fujaba Models with ATL

For EMF, there are plenty of model transformation and management tools available; many of them in the Eclipse Modelling Project [11]. As one representative, we show the ATL [1] transformation tool, in which model transformations can be defined in declarative rules. One could for instance define a transformation from Fujaba UML to Eclipse UML2 [2] and then open the result with editors for the latter. Figure 4 shows an excerpt of such a transformation and the result of transforming the conference system model.

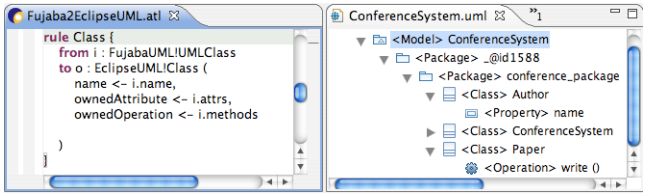


Figure 4: Transforming Fujaba to Eclipse UML

3.4 Reuseware: Composing Fujaba Models

In our own tool Reuseware [6, 10] we can define (cross-cutting) composition systems for modelling languages. We defined, for instance, a composition system for Fujaba class diagrams that can be used to extend classes with pre-defined functionality. Figure 5b shows a Reuseware composition program that extends the example model with observer behaviour by reusing the prior defined observer model (cf. Figure 5a). Reuseware can execute the composition program and produce a composed model (cf. Figure 5c). This application was indeed our motivation for developing the adaptation mechanism and will be further explored in future research.

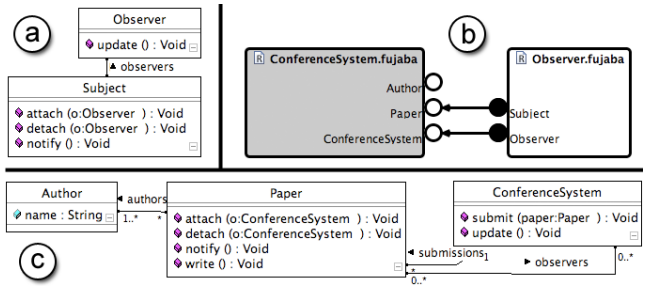


Figure 5: Composing two Fujaba models

4. CONCLUSION AND OUTLOOK

In this paper we described a metamodel mapping and a set of adapters that enable EMF tools to access models in the Fujaba workspace. This opens the door for interesting new projects that combine EMF tools and Fujaba.

A prerequisite for the adaptation is the availability of an Ecore version of the Fujaba metamodel. Since this was not available, a tool was written to extract this metamodel from Fujaba's class files. This tool requires additional information, which are at the moment statically encoded. In the future, this could be made configurable, and different configurations could be provided for different Fujaba versions to ensure that the tool works with them. Another approach would be to construct a complete Fujaba metamodel in UML by using Fujaba reverse engineering tools and, if required, manual modelling. [5] can then be applied to transform it into an Ecore metamodel of Fujaba.

We realised dynamic adapters using reflection. One could also think of a generative approach that generates specific adapters for metaclasses, avoiding the usage of reflection to increase performance. Such an approach might also be applicable the other way around—to access EMF models from Fujaba. For this, Ecore metamodels would have to be mapped to extensions of Fujaba's metamodel and adapters that work in the reverse direction must be generated.

For the adapters to work, both the EMF tool and Fujaba have to run in the same JavaVM, which caused some ClassLoader issues during our experimentation. How to resolve those has to be further investigated. Another open issue is to integrate the adapters with Fujaba4Eclipse, which should not be difficult, since Fujaba4Eclipse is based on the same framework as Fujaba. On the other hand, the integration should be much smoother and less problems should occur, since Fujaba4Eclipse and the EMF-based tools both run inside Eclipse (and therefore in the same JavaVM).

5. ACKNOWLEDGMENTS

This research has been co-funded by the European Commission within the 6^{th} Framework Programme project Modelplex contract number 034081 (cf. `www.modelplex.org`).

6. REFERENCES

[1] ATL Project. Atlas Transformation Language. `www.eclipse.org/m2m/atl`. Accessed Aug. 2008.

[2] Eclipse Foundation. Eclipse UML2 Project. `www.eclipse.org/uml2`. Accessed Aug. 2008.

[3] EMF Compare Project. Emf compare. `www.eclipse.org/emft/projects/compare`. Accessed Aug. 2008.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[5] L. Geige, T. Buchmann, and A. Dotor. Emf code generation with fujaba. In L. Geiger, H. Giese, and A. Zündorf, editors, *Proc. of the 5^{th} International Fujaba Days, Kassel, Germany*. University of Kassel, Oct. 2007.

[6] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On language-independent model modularisation. *Transactions on Aspect-Oriented Development, Special Issue on Aspects and MDE (to appear)*, Nov. 2008.

[7] F. Heidenreich and U. Wemmie. Breaking the domination of the internal graph model. In L. Geiger, H. Giese, and A. Zündorf, editors, *Proc. of the 5^{th} International Fujaba Days, Kassel, Germany*. University of Kassel, Oct. 2007.

[8] J. Johannes, I. Savga, and T. Haupt. Integrating fujaba and the eclipse modeling framework. In H. Giese and B. Westfechtel, editors, *Proc. of the 4^{th} International Fujaba Days, Bayreuth, Germany*, volume tr-ri-06-275. University of Paderborn, Sept. 2006.

[9] Object Management Group. MetaObject Facility (MOF) specification version 2.0. OMG Document, Jan. 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.

[10] Reuseware Project. Reuseware Composition Framework. `http://www.reuseware.org`. Accessed Aug. 2008.

[11] The Eclipse Foundation. Eclipse modelling project. `http://www.eclipse.org/modelling`. Accessed Aug. 2008.

The Fujaba Automotive Tool Suite*

Kahtan Alhawash, Toni Ceylan, Tobias Eckardt, Masud Fazal-Baqaie, Joel Greenyer, Christian Heinzemann, Stefan Henkler, Renate Ristov, Dietrich Travkin, Coni Yalcin
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[alhawash|crowdy|tobie|masudf|jgreen|chris227|shenkler|renate|travkin|coni81]@upb.de

ABSTRACT

Automotive systems contain a large number of software controllers that interact in order to realize an increasing number of functions. The controllers are typically developed separately by different suppliers. Therefore, errors in the overall functionality are often detected late in the development process or even remain undetected. This affects the quality and safety of the systems and may lead to expensive recalls.

We propose to specify the communication behavior more precisely in the early system design by modeling the controllers' interactions using formal sequence diagrams (LSCs). Based on these behavior models we're able to automatically synthesize state machines, which can be used to generate code for the communication behavior of the controllers. We provide a prototypical integrated modeling environment based on Fujaba which supports specifying requirements, modeling the component architecture, and component behavior as well as the state machine synthesis.

1. INTRODUCTION

Today's cars provide a continuously growing number of enhanced functions like a car's windows when it starts raining and no one is in the car, or automatically adapting the speed to that of a preceding car. The rising complexity of these functions requires the interaction of numerous controllers, sensors, and actuators to achieve the desired behavior. In order to handle the system's complexity and increase its flexibility, more and more functions are realized by software.

The controllers are usually developed separately by different suppliers. These rely on interface and requirements specifications given by car manufacturers. Therefore, the controllers are tested as single units. Integration tests, especially tests addressing communication behavior, are run very late in the development process. Not sufficiently considering the communication behavior in the early development process, negatively affects the system's quality and safety. Failures and errors due to wrong communication behavior are revealed too late and may result in product recalls.

Furthermore, inconsistencies in system design may arise due to heterogenous tools which are typically used in the automotive industry today: DOORS[1] is often used for requirements engineering, an AUTOSAR[2]-compatible editor is used to specify the system architecture, and MATLAB Simulink[3] for the specification of the components' internal controller behavior. Finally, code generators like TargetLink[4] are used to generate the controller code.

We approach the problems stated above by formally specifying the communication behavior earlier in the development process, starting in the requirements engineering phase. The requirements are refined step-wise to iteratively build the system architecture and formally specify its behavior very early in the system design. This would enable car manufacturer to validate the overall system functionality in earlier development phases and to provide suppliers with more precise specifications of single controllers. Additionally, we show how to establish traceability between the design artifacts and the elaborated requirements. To support the overall process, we implemented a modeling environment based on Eclipse and Fujaba. Since all editors operate on the same underlying model and provide just different views on it, consistency between all artifacts and across all development phases is achieved.

2. EXAMPLE

The example that we use throughout this paper is that of an adaptive cruise control (ACC). The ACC supports the driver of a car by keeping the current speed and by adapting the speed to preceding cars automatically in order to maintain a safe distance.

A simplified system architecture for the ACC is shown in Figure 1. The sensors, actuators, and the user interface are omitted here. The ObjectRecognition component is responsible for detecting preceding cars by interpreting inputs from a front radar sensor. The information about preceding cars is sent to the AdaptiveControl component, which computes the maximum speed of the car so that the safety distance is maintained. The new speed of the car is set in the AccelerationControl component, which is responsible for controlling the actual speed of the car. The internal (controller) behavior, e.g. the calculation of the new speed by the AdaptiveControl, is not considered here. These are typical tasks for control engineering, where control behavior is specified by continuous block diagrams, usually using MATLAB Simulink.

However, the three components have to interact in order

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

[1]http://www.telelogic.com

[2]http://www.autosar.org

[3]http://www.mathworks.com

[4]http://www.dspace.com

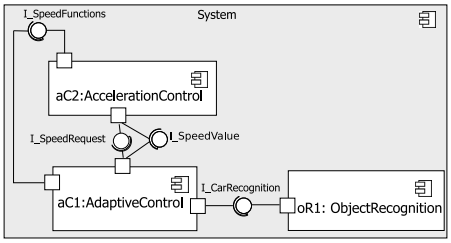


Figure 1: Simplified ACC component diagram

to achieve the correct ACC behavior. To achieve a correct system functionality, a correct communication behavior of the components is needed in addition to the internal behavior. In this paper, we concentrate on the communication behavior, which is modeled in terms of asynchronous messages that are sent via ports.

3. THE DEVELOPMENT PROCESS

The development process in the automotive industry today captures the system's requirements in a structured way, typically using the tool DOORS. In order to ease integration of structured requirement specifications into our tool environment, we specify requirements in a goal-oriented way [6], using goal trees (Figure 2). This helps to iteratively define requirements, identify components and describe communication behavior. These goals are therefore directly linked to the according parts in the system architecture (components) and communication behavior descriptions.

The system architecture is modeled using UML component diagrams (see Figure 1). Component diagrams have become the standard means to specify the architecture of automotive systems today, since a variant of component diagrams also forms part of the AUTOSAR standard. Components can send and receive messages via ports. The ports are typed over interfaces which contain a set of message types that can be sent or received (Figure 3).

In the early design process found in practice today, it is specified what kind of data values or messages can be communicated between the components, but the order of messages and the events which trigger communication sequences are not formally specified. Thus, the behavioral specification given to suppliers is often incomplete. Different from industry processes, we propose to specify communication between components in the early design phase by using a formal variant sequence diagrams (see section 5). This enables an early analysis of the communication behavior before the specification is given to suppliers. In our implemented automotive tool suite, the sequence diagrams are modeled based on the system architecture model and thus syntactic consistency is ensured. Furthermore, we link components and sequence diagrams to the goals which they fulfill so that design problems can be traced to the according requirements.

All parts of the system description are iteratively refined during the specification process. The communication behavior for each component can then be synthesized automatically. Our synthesis generates one statechart per component. From these statecharts, source code can be generated by standard techniques [2]. This code can be used for early prototyping, or code generation may be combined with the refined (control) behavior of the component, which is developed by component suppliers [1].

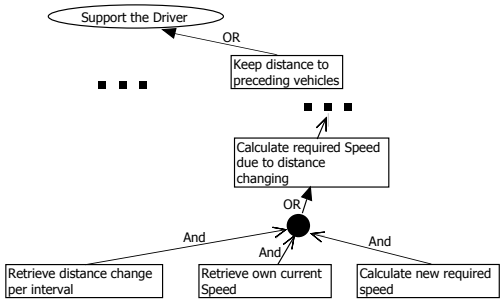


Figure 2: Goal tree excerpt of the ACC example

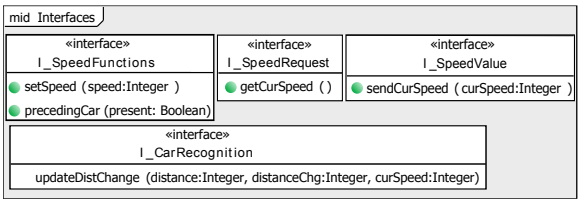


Figure 3: Interfaces – Specifying communication between components

4. MODELING COMMUNICATION BEHAVIOR

In this section, we show how to specify the communication behavior by considering two refined goals from the example "Retrieve own current speed" and "Calculate new required speed" (Figure 2).

This communication behavior is described with a subset of Live Sequence Charts (LSCs) [5]. The syntax, however, is based on UML 2.1.

We specify a separate LSC for each goal – Figure 4a refines the behavior for "Retrieve own current speed", Figure 4b refines the behavior for "Calculate new required speed".

According to [5], we distinguish between mandatory behavior and possible scenarios. To describe what must happen, we split sequence diagrams into a prechart and a main chart. If the behavior of the prechart is observed, the behavior of the main chart is executed. The LSC in Figure 4a accordingly specifies that if the AccelerationControl receives getCurSpeed() from the AdaptiveControl, it will answer with sendCurSpeed(curSpeed). The LSC in Figure 4b specifies that if the current speed has been retrieved, the AdaptiveControl has to calculate the new speed by calling calcMaxSpeed(...) and send it to the AccelerationControl using setSpeed(...). Specifying system behavior in such manner thus enables early execution and analysis of specifications.

Several LSCs can be active at the same time. Since the execution of one LSC can lead to the fulfillment of the prechart of another, one LSC execution can be a trigger for another LSC execution. This way, the execution of the main chart in Figure 4a triggers the LSC in Figure 4b.

Further on, we distinguish between synchronized method calls for intra-component behavior (calcMaxSpeed(...)) and asynchronous messages for inter-component communication.

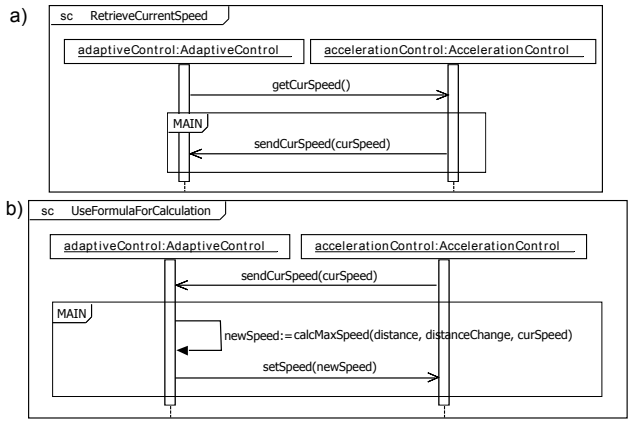


Figure 4: a) Sequence diagram for speed retrieval b) Sequence diagram for calculation of new speed

The central advantage of using LSCs is their formality, which enables synthesis to one statechart for each component. Additionally, the specification can already be simulated and analyzed during the requirement specification phase. This solves one of the major problems of the current development process employed in industry where incorrect communication behavior is often discovered not until implementation phase.

5. SYNTHESIS OF COMMUNICATION BEHAVIOR

Based on the system's inter-component communication behavior specified by a number of LSCs, we are now able to automatically synthesize intra-component behavior for each component as statecharts similar to the methods proposed in [3] and [4]. We adapted the implementation of [3] to synthesize LSCs [4].

We generate one statechart for each component with one orthogonal state for each LSC. The top orthogonal state of the statechart for the AdaptiveControl (Figure 5) results from the LSC RetrieveCurrentSpeed, the bottom orthogonal state results from the LSC UseFormulaForCalculation (Figures 4a and b).

For generating the states for each LSC, we insert states on the component's lifeline before and after each sending or receiving of a message. The transition between those states becomes either a raised – sending a message – or a trigger event – receiving a message – with the message as the transition label. In the example, two states InitReceivesendCurSpeed and ReceivedsendCurSpeed are generated for the reception of sendCurSpeed(...). Furthermore, a trigger sendCurSpeed(curSpeed) is added to the transition between those states.

If the message is located in the prechart of the sequence diagram, the event is always a trigger, whether the message is sent or received. The reason for this is that the prechart is observed behavior and not executed behavior. In the example, the sent message getCurSpeed() is a trigger of the corresponding statechart transition between the states InitiategetCurSpeed and SentgetCurSpeed.

Intra-component method calls are realized as entry actions in one state. Accordingly for the method calcMaxSpeed(...) a state IncalcMaxSpeed is generated with the method call as its entry action.

To realize correct behavior for the main chart semantics, coordination messages are employed as proposed in [4]. To give notice to all participating components that the prechart of the LSC is satisfied, a message is sent to the corresponding components. As soon as those receive such a message, they change their state for being able to react on or execute the ensured behavior of the main chart. In the example statechart (top orthogonal state), the AdaptiveControl first has to receive coordadaptiveControlgetCurSpeed() before it starts with the execution of the main chart. Accordingly, coordination messages have to be inserted to notify all participating components that the execution of the main chart has finished (message overaccelerationControl()).

This automatic synthesis of the components' communication behavior facilitates analysis of communication based component and system properties at an early stage of development. It also enables automatic code generation. For this, the intra-component method calls have to be extended with internal controller behavior.

6. RELATED WORK

Our sequence diagrams are based on LSCs [5], using a synthesis sceme as presented later by Harel, Kugler and Pnueli [4]. The main difference of our subset of LSCs, besides the UML syntax, is that they only employ hot semantics and only describe conditions within alternative and loop frames. In addition to [4], we are able to synthesize these alternative and loop frames. The original work on LSCs relies on a simple object system to describe the system architecture, whereas we consider a component-based software architecture. Our synthesis is designed to route coordination messages (see section 5) only over existing connectors between the components instead of allowing arbitrary communication between the components. This becomes important when deploying the components on a distributed platform.

The SceBaSy approach uses time-annotated sequence diagrams to synthesize reusable, state-based coordination patterns [3]. The synthesis focuses on calculating admissible time bounds for real-time statecharts, but in turn requires the sequence diagrams to be deterministic. This requires a detailed behavior description and, therefore, the approach is not suitable for early development phases. SceBaSy and its real-time analysis could, however, still be integrated in a later development stage. To integrate SceBaSy, it should be investigated in the future how to perform time-analysis to find admissible alternatives in early, non-deterministic communication behavior descriptions. Furthermore, the SceBaSy synthesis employs rules for optimizing the resulting statecharts and it is to be investigated how to employ such rules in our synthesis scheme.

7. CONCLUSION AND FUTURE WORK

We presented a development process for automotive software systems with special emphasis on formal specification of communication behavior. We proposed a sequence diagram dialect that facilitates early analysis of communication

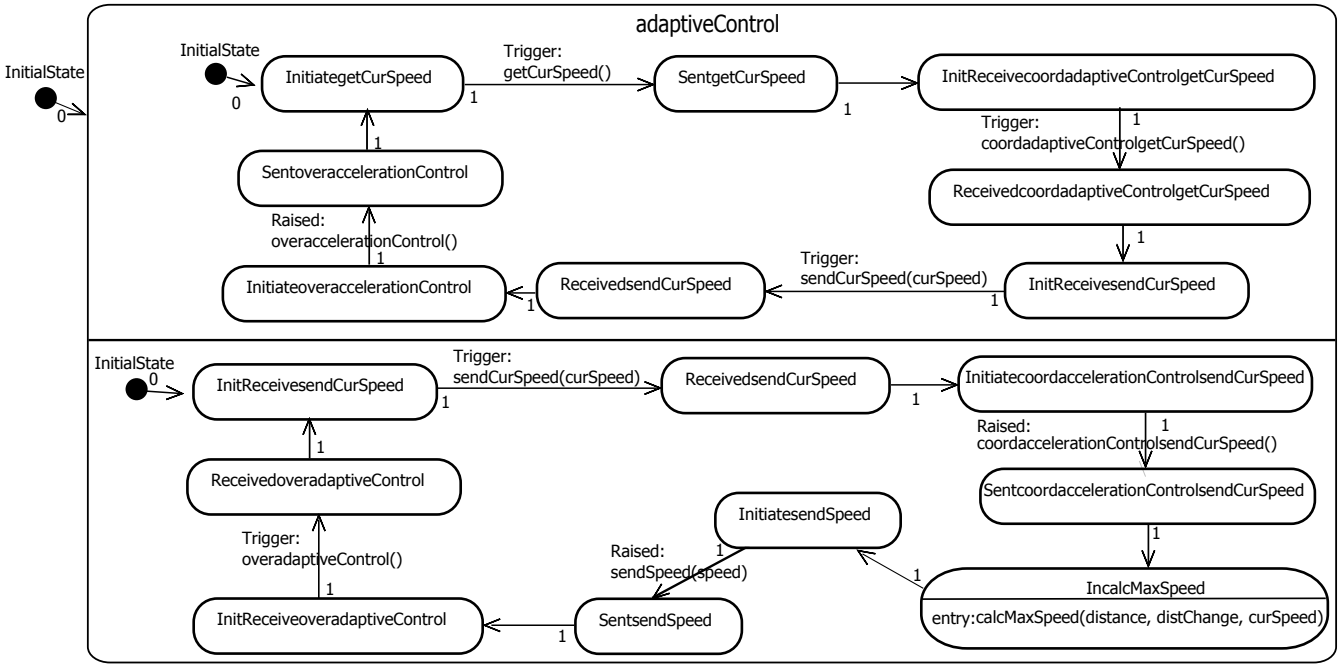


Figure 5: Synthesised statechart for the adaptiveControl component

behavior and automatic synthesis to communication based component behavior by means of statecharts. We realized tool support for this development process and proposed concepts using the Fujaba4Eclipse platform.

Future work embraces scalablity analysis of the synthesized statecharts. We suppose that, as the approach uses parallel states, it provides better scalability for statecharts than synthesizing one non-parallel statechart for each component. Accordingly, synthesized statecharts should be optimized to eliminate superfluous states.

Also important for future work are consistency checks on LSCs, for example showing deadlock freedom or discovering contradictions between several LSCs. This is important, because a high modularity in the LSC specification can lead to many inconsistency that are not obvious to see.

It is already possible to annotate time constraints in our LSCs. The synthesis approach needs to be extended with timing analysis as the specified systems act in real-time environments. One possibility could be to integrate SceBaSy for use in later design phases of the overall development process while using our approach in the requirement specification. Controller behavior has to be integrated for example using MATLAB Simulink models. Another extension could be the inclusion of a deployment model (cf. AUTOSAR) in order to specify which software components shall be realized on which hardware components. Finally, an AUTOSAR conform code generation should be developed to support the implementation phase of the automotive software system.

8. REFERENCES

[1] S. Burmester, H. Giese, S. Henkler, M. Hirsch, M. Tichy, A. Gambuzza, E. Müch, and H. 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.

[2] S. Burmester, H. Giese, and W. Schäfer. Code generation for hard real-time systems from real-time statecharts. Technical Report tr-ri-03-244, University of Paderborn, Paderborn, Germany, October 2003.

[3] H. Giese, S. Henkler, M. Hirsch, and F. Klein. Nobody's perfect: interactive synthesis from parametrized real-time scenarios. In *SCESM '06: Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 67–74, New York, NY, USA, 2006. ACM.

[4] D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling*, pages 309–324, Berlin/Heidelberg, Germany, 2005. Springer-Verlag.

[5] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[6] A. V. Lamsweerde. Goal-oriented requirements engineering: A guided tour. *RE '01: Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, page 249, 2001.

Hybrid Model Checking with the FUJABA Real-Time Tool Suite*

Stefan Henkler, Martin Hirsch, Claudia Priesterjahn
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[shenkler|mahirsch|cpr]@uni-paderborn.de

ABSTRACT

Advanced mechatronic systems use their software to exploit local and global networking capabilities to enhance their functionality and to adapt their local behavior. Such systems include complex hard real-time coordination at the network level. This coordination is further reflected locally by complex reconfiguration in form of mode management and control algorithms. As such hybrid systems often contain safety-critical requirements, a proper approach for the safety analysis is mandatory. In former papers we have presented a compositional verification approach for the real-time and safety analysis. We present in this paper the integration of the hybrid verification tool PHAVer in the FUJABA Real-Time Tool Suite which enables also to consider hybrid requirements.

1. INTRODUCTION

For mechatronic systems [2], which have to be developed in a joint effort by teams of mechanical engineers, electrical engineers, and software engineers, the advances in networking and processing power provide many opportunities. The development of such systems will therefore at first require means to develop software for the complex hard real-time coordination of its subsystems at the network level. Secondly, software for the complex reconfiguration of the local behavior in form of mode management and control algorithms is required, which has to proper coordinate the local reconfiguration with the coordination at the network level.

The interplay between the discrete software models and the continuous controllers is the cause for hybrid requirements. As such systems often contain safety-critical requirements, a proper approach for the safety and hybrid analysis is mandatory.

In [3] and [5] we have presented the FUJABA Real-Time Tool Suite which enables the model-based development of mechatronic Systems as well as the formal verification of the real-time coordination and the correct embedding of the employed controllers. But, the current Tool Suite lacks in the support for analyzing hybrid behavior.

We present in this paper the integration of the hybrid verification tool PHAVer [6] into the FUJABA Real-Time Tool Suite based on the mappings from Real-Time Statecharts to hierachical Timed Automata [7, 8]. In detail, we have to map Hybrid Reconfiguration Charts (HRC), our modeling approach for hybrid systems, to Hybrid Input/Output Automata (HIOA) [6], the input model of PHAVer.

In the remainder of this paper, we at first present in Section 2 the conceptual integration of the PHAVer tool. Then, we outline in Section 3 the tool support and evaluation. Finally, we conclude and present future work.

*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.

2. HYBRID MODEL CHECKING OF UML MODELS

In this section the concept of the integration of the hybrid model checker PHAVer and the necessary mapping rules are presented. Figure 1 shows the actions that have to be performed for the model checking of HRCs with PHAVer. The main step is the transformation from the HRC model into a HIOA. The HIOA and a specification to be checked build the input for PHAVer. The model checker states if the model fulfills the specification or not. In order that the verification

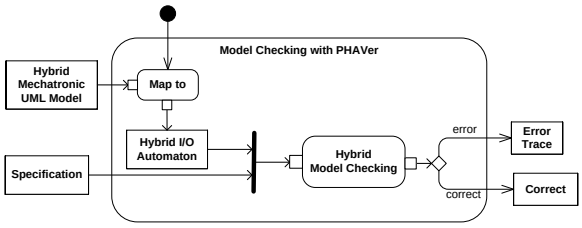


Figure 1: Hybrid Model Checking with PHAVer

will perform correctly the mapping must keep the semantics of the original model. Otherwise we might find false positives or false negatives during verification. The following paragraphs introduce the mapping rules established in this work.

Start State.

A start state in a HRC can embed configurations of the component. The continuous values at the component's ports can thus be deployed in this state. However, in contrast to HIOA reconfiguration charts can not specify their inital values or range of values. The start state of the HRC is mapped to the start state of the HIOA consisting of a location and variable allocations. Each continuous variable is assigned the value 0 (Figure 2).

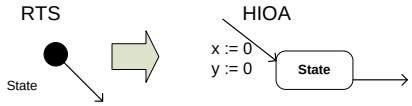


Figure 2: Mapping a start state

Clocks and Clock Resets.

Each clock t_k of the original timed automaton is mapped as a continuous variable t_k such that for each location of the HIOA $\dot{t}_k = 1$ is satisfied.

Further in HIOA discrete assignments can be assigned to transitions. Therefore a clock reset can simply be modeled by assigning 0 to the clock to be reset (Figure 3).

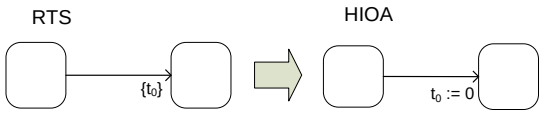


Figure 3: Mapping clock resets

Time Invariants and Time Guards.

Because of the direct mapping of clocks to continuous variables both time invariants and time guards can directly be taken over to the HIOA (Figure 4).

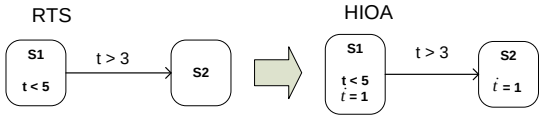


Figure 4: Mapping guards

Urgent and Non-Urgent Transitions.

As HIOA allow only urgent transitions, all non-urgent transitions of the hybrid reconfiguration chart are mapped to urgent-transitions. Thus the original semantics are kept since "non-urgent" says that a transitions does not need to fire immediately after its activation. In case such a transition would actually fire immediately no unexpected behavior would result [12].

Synchronous Communication.

The mapping for synchronous communication is depicted in Figures 5 and 6. Figure 5 shows the synchronization modeled by a HRC. Parallel states of hybrid reconfiguration charts synchronize via synchronization channels. Each synchronization contains exactly one sender and one receiver. At the starting point the HRCs as depicted in Figure 5 start with states $S1$ and $S3$. When the event e is activated the statecharts switch to $S2$ and 4 synchronously.

Figure 6 shows the same synchronization modeled by two HIOA. Generally transitions of two parallel HIOAs will fire simultaneously, if they are labeled with the same synchronization marks and are activated at the current moment. $S1$ is the source of the transitions sending messages. That is why all incident transitions have to contain the variable *e_send* representing the sender. For all incoming transitions *e_send* is set to 1. All outgoing transitions must reset this value to 0 - the initial value. The receiving transition

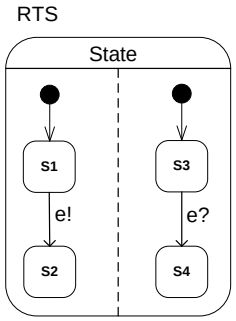


Figure 5: Synchronous communication modeled by hybrid reconfiguration chart

$(S3, S4)$ will only be fired, if *e_send* is set to 1 and another transition also marked with the label e is activated. Thus sender and receiver can only fire synchronously and the original semantics of synchronous communication in HRCs are kept.

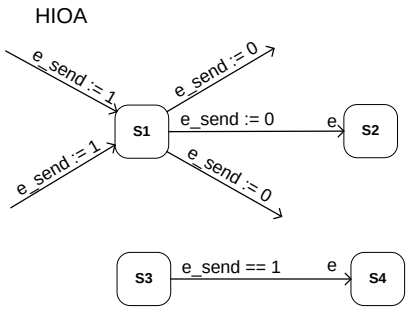


Figure 6: Synchronous communication modeled by HIOA

Embedding of Component Instances.

In HRCs the behavior of controllers is described by differential equations over the incoming and outgoing signals. The same applies to the component instances embedded in states of hybrid reconfiguration charts. As the continuous dynamics of hybrid systems in PHAVer is also described by differential equations they can be directly applied to HIOA.

Application Order.

The existing XML exchange format for the transformation of Mechatronic UML models to the UPPAAL input format [8] has been extended by continuous parts to enable the verification of continuous parts of mechatronic systems. Also the transformation of Real-Time Statecharts into hierarchical Timed Automata has been extended by continuous parts. In that way the following subset of the needed mappings can already be performed: stop states, entry()-, do() and exit()-methods and the history operator.

Afterwards the mapping rules presented in this paper are applied. To guarantee the correctness of the resulting models the mapping steps have to be applied in a defined order. For instance the mapping rule for clocks demands that all locations of the resulting model satisfy a certain property. Therefore it must not happen that one of following mapping steps adds a location that does not satisfy this property.

The following list shows the appropriate application order of the transformation steps.

1. First apply the rules presented in [8].
2. Map the synchronous communication.
3. Map the non-urgent transitions, since following mappings can not add new locations or transitions to the model.
4. Map the clocks.
5. Map time guards, time invariants and clock resets.

3. TOOL SUPPORT AND EVALUATION

In this section we explain the implementation of the approach. First, we describe the implementation as a plugin for FUJABA. Thereafter we give an evaluation example and point out the adaptability of the verification to real world examples.

3.1 The Plugin

In Figure 7 the architecture of the required FUJABA plugins is depicted. The three plugins HybridComponenent, UMLRT2, and RealtimeStatechart allows us to model the behavior as well as the structure of hybrid systems. The plugins on the bottom left side realize the integration of model checker. The UMLRTModelchecking provides an interface for model checker which allows to export the model and add constraints to the model [4]. The actual model checking engine is provide by the specific model checker, in our case the PHAVer-Plugin.

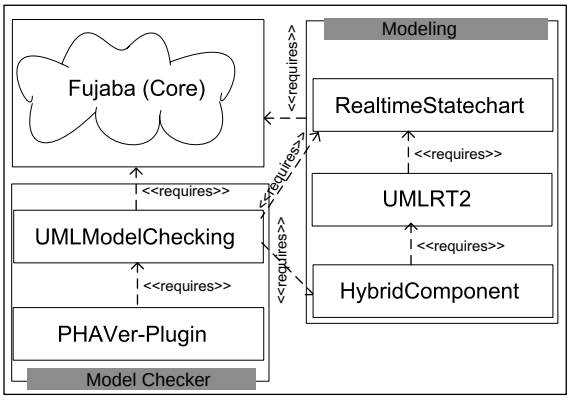


Figure 7: Architecture of the plugin

Figure 8 shows the transformation steps taken by the implemented FUJABA plugin. First the hybrid reconfiguration chart is exported to the XML exchange format provided by the UMLModelChecking Plugin. This action is followed by the transformation into an intermediate format and the transformation into the final PHAVer input format the hybrid input output automata.

3.2 Evaluation

As an evaluation example we take the AHCS case study from [10]. We consider the model of a central controller for a automated highway and analyze the controller itself for safety properties, particularly for the specification that

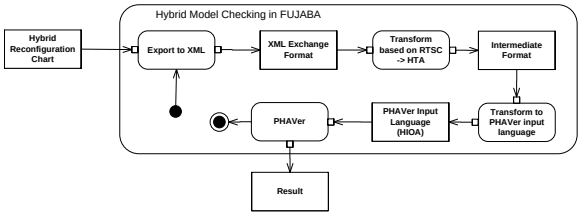


Figure 8: Hybrid Model Checking in FUJABA

no two vehicles on the automated highway collide with each other. The controller enforces speed limits on vehicles on the automated highway to achieve this purpose. In Figure 9 the hybrid reconfiguration chart to which realizes the behavior of the AHCS for four vehicles is depicted. x_k is the distance of one vehicle k to the beginning of the highway and $\dot{x}_k$ the velocity of the vehicle. At the beginning (state "'Fahrt"') the velocity for each vehicle is in the interval $\dot{x} \in [a, b]$. When two vehicles i, j come within a distance α ($x_i - x_j < \alpha$) of each other, we call this a "'possible"' collision event and the controller switches to the "'Risiko_i_j"' state. The controller asks the approaching vehicle to slow down by reducing the upper bound to $\dot{x}_i \in [a, c']$ and asks the leading vehicles to speed up by increasing the lower bound to $\dot{x}_j \in [c, b], c > c'$; it also requires that all other cars not involved in the possible collision slow down to a constant velocity β for vehicles behind the critical region and $\beta', \beta' > \beta$ for vehicles in front of the critical region. When the distance between the two vehicles involved in the possible collision exceeds α, the controller switches to the "'Fahrt"' state . Otherwise the "'Error"' state will be reached, if $x_j - x_i < \alpha', alpha' < \alpha$

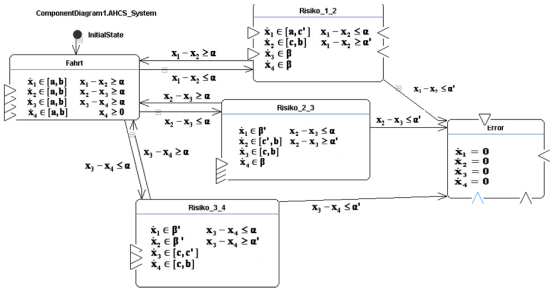


Figure 9: Evaluation example

To ensure that no collision between the vehicles happen we have to check that the "'Error"' state will be never reached as long as the systems runs. To check this constraint we have to compute the followings steps in PHAVer:

- compute all reachable state: Reach = System.reachable
- define the forbidden states: Forbidden = System.{Error? & True}
- compute the set $Reach \cap Forbidden$: bad.intersection_assign(Forbidden)

In Figure 10 a screenshot of the constraint is depicted. For the evaluation of the example, we chose the following parameterization of the variables. WE set the number of vehicles

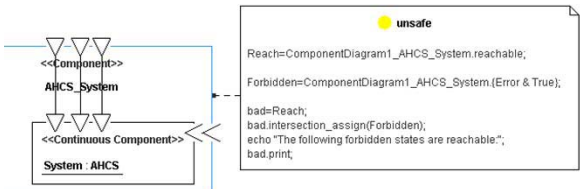


Figure 10: Specification of the constraint

to 4, 6, 8, 12, 14, and 16. The other variables are set to:

$a = 0, b = 100, c = 70, c' = 50, \alpha = 100, \alpha' = 10, \beta = 40, \beta' = 80$

.

The results of the evaluation are presented in Table 3.2 as well as a visualization of the results is given in 11. In detail the time and memory consumption are analyzed. One result is that the runtime of the model checker exponentially increase with the number of vehicles. The same holds for the memory consumption. It is to be noted that although PHAVer is more efficient than other model checkers [6] the runtime exponentially increase with the number of variables used in the HIOAs [11]. Hence it can take a long time to get a result if any result is computed.

Number of vehicles	Runtime [s]	Memory [MB]
4	0, 16	3 MB
8	2, 6	11 MB
10	11, 34	16 MB
12	156, 7	37 MB
14	5120, 3	112 MB
16	14092, 34	354 MB

Table 1: Evaluation results

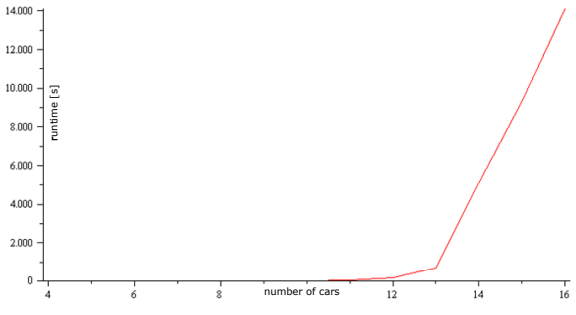


Figure 11: Evaluation results

4. CONCLUSION AND FUTURE WORK

In this paper, we have presented the integration of the hybrid verification tool PHAVer in the FUJABA Real-Time Tool Suite. We have shown the mapping of our hybrid modeling approach, Hybrid Reconfiguration Charts, to the input model of PHAVer (Hybrid Input/Output Automata). In the evaluation, we have shown that, in principle, the model checking of hybrid systems does not scale. Therefore, appropriate abstractions are required as shown in [9]. In the future, we want to consider a better integration of the PHAVer tool in the compositional verification approach by using the assume/guarantee approach of PHAVer [1] which enables a better scalability.

5. REFERENCES

[1] S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. *Lecture Notes in Computer Science*, 1536:81–102, 1998.

[2] D. Bradley, D. Seward, D. Dawson, and S. Burge. *Mechatronics*. Stanley Thornes, 2000.

[3] S. Burmester, H. Giese, S. Henkler, M. Hirsch, M. Tichy, A. Gambuzza, E. Müch, and H. 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.

[4] S. Burmester, H. Giese, M. Hirsch, and D. Schilling. Incremental design and formal verification with UML/RT in the FUJABA real-time tool suite. In *Proceedings of the International Workshop on Specification and vaildation of UML models for Real Time and embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, October 2004. to appear.

[5] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The fujaba real-time tool suite: Model-driven development of safety-critical, real-time systems. In *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, pages 670–671. ACM Press, May 2005.

[6] G. Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. pages 258–273. Springer, 2005.

[7] H. Giese and S. Burmester. Real-Time Statechart Semantics. Technical Report tr-ri-03-239, University of Paderborn, Paderborn, Germany, June 2003.

[8] M. Hirsch. Effizientes Model Checking von UML-RT Modellen und Realtime Statecharts mit UPPAAL. Master's thesis, University of Paderborn, June 2004.

[9] M. Hirsch, S. Henkler, and H. 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 1–8. ACM Press, May 2008. to appear.

[10] S. K. Jha, B. H. Krogh, J. E. Weimer, and E. M. Clarke. Reachability for linear hybrid automata using iterative relaxation abstraction. In A. Bemporad, A. Bicchi, and G. C. Buttazzo, editors, *HSCC*, volume 4416 of *Lecture Notes in Computer Science*, pages 287–300. Springer, 2007.

[11] X. Li, S. J. Aanand, and L. Bu. Towards an efficient path-oriented tool for bounded reachability analysis of linear hybrid systems using linear programming. *Electron. Notes Theor. Comput. Sci.*, 174(3):57–70, 2007.

[12] A. Steinke. Integration Hybrider Rekonfigurationscharts mit Matlab/Simulink-Modellen. Master's thesis, University of Paderborn, 2007.

Component Story Diagrams in Fujaba4Eclipse*

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

ABSTRACT

A current trend in Software Engineering is the development of self-adaptive component-based software architectures. Our modeling approach for component-based software systems MECHATRONIC UML, integrated in the FUJABA Real-Time Tool Suite plugin for Fujaba4Eclipse, supports self-adaptiveness only to a certain degree. This paper presents tool support for an extension of MECHATRONIC UML, which facilitates initialization and reconfiguration of a MECHATRONIC UML system based on Story Diagrams and thus enables a step towards self-adaptiveness on a structural level.

1. INTRODUCTION

Today's software systems are mainly build in a component-based fashion to reduce their complexity. Furthermore, some software systems have to be self-adaptive, which means that they adapt their behavior in response to event occurrences or changes in the environment. The adaption can be realized by changing system parameters or by a structural reconfiguration. MECHATRONIC UML [1] supports, among other things, the modeling of components with real-time behavior, the coordination between distributed components as well as its verification, but only a limited structural reconfiguration by enumerating all different configurations.

In [6], we presented concepts for an extension to MECHATRONIC UML, which overcomes this limitation. This paper presents the implementation of these concepts in Fujaba4Eclipse. The general idea is to use a formalism based on Story Diagrams [2] to model system initialization and reconfiguration for a component-based architecture. Component instance structures consist in contrast to object structures of elements such as component and port instances and different connector types. Furthermore, conventional Story Diagrams operate only on flat object structures and do not provide the possibility to traverse a hierarchy, which is induced by a component-based system. So traditional Story Diagrams are not well suited for component-based architectures, but their variant *Component Story Diagrams* [6] make use of both the Story Diagrams' formal foundations and sophisticated control structures as well as the extensions required by component-based architectures. The developed formalism reuses the graphical syntax of component diagrams for a tight integration of the modeling of structure as well as reconfiguration behavior. We refer to [3, 6] for a discussion of the limitations of related approaches.

The running example used in this paper is taken from [6] and describes the communication structure of autonomous railway vehicles from the RailCab project[1]. These RailCabs can build convoys to reduce the energy consumption by utilizing the slipstream. Their communication structure is build by MECHATRONIC UML components. The leading RailCab computes reference positions for all subsequent RailCabs in a convoy. For each following RailCab, one position calculation component is deployed in the leading RailCab. This component sends the reference position to its connected following RailCab. Thus, a structural reconfiguration is required for the leading RailCab's embedded components when a RailCab joins or leaves a convoy.

Section 2 sketches a flexible component type definition as a base for the modeling of Component Story Diagrams. While Section 3 presents the Component Story Diagrams themselves, Section 4 introduces the code generation plugin, which is a prerequisite for the execution of the Component Story Diagrams. Section 5 concludes this paper and adduces current and future work on Component Story Diagrams.

2. COMPONENT TYPE DEFINITION

Like conventional Story Diagrams, also Component Story Diagrams require a type definition as a basis for modeling. Instead of classes, component types are employed as classifying model. In [6] we presented a variable component type definition based on the UML Composite Structures [4]. Component types consist of *parts*, which are also classified by a component type and have a multiplicity. So parts represent a set of instances classified by the part's component type, which an instance of the superordinate component type may contain by composition. Additionally, ports are provided with a multiplicity.

Figure 1 shows the embedded component structure for the component type RailCab, holding the behavior concerning computation of reference positions for subsequent RailCabs and adaptation of the own position to a reference position. Besides some parts with a default multiplicity of 0..1 for component instances which control and determine the position or the velocity of a RailCab, the part posCalc—classified by the component type PosCalc which contains the behavior for calculating reference positions for following RailCabs—is deployed within RailCab. This part has a multiplicity of

*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] www.railcab.de/en

$*$ (denoted by the second frame), so it stands for an arbitrary amount of embedded component instances of the type PosCalc. One of its regular ports is connected via a delegation connector type PosRef to a so-called *multiport* with multiplicity $*$ attached to the superordinate component type RailCab. Instances of this multiport are used for connections between the embedded position calculation component instances with component instances representing the following RailCabs. Since the delegation connector type PosRef has a source and target multiplicity of 1, each component instance of the part posCalc may be connected via a delegation connector :PosRef exactly to one instance of the corresponding multiport. In the same manner, component instances of the part posCalc may be connected among each other via assembly connectors :Next. That way, a position calculation component instance gets the reference position of the preceding RailCab as input for the calculation of the reference position of the subsequent RailCab.

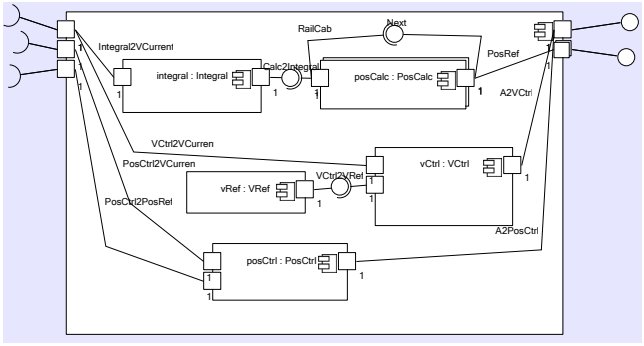


Figure 1: Component type RailCab [3]

Figure 2 depicts the system level ConvoySystem, which describes all allowed system configurations with interconnected RailCab parts: the assembly connector type PosRef connects the afore mentioned multiport of the part convoyLeader with multiplicity 0..1 to a regular port of the part convoyFollowers with multiplicity of $*$. This means that there is at most one instance of the role convoyLeader, which is connected by n :PosRef connectors to n convoyFollowers instances. In this way, the latter ones can receive the reference positions calculated by the corresponding posCalc instances, which are deployed within convoyLeader. Together with the internal definition of the component type RailCab, this builds the component-based communication structure for RailCab convoys.

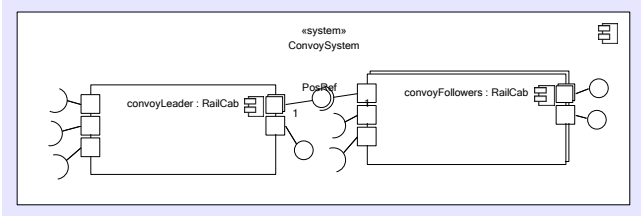


Figure 2: System level ConvoySystem [3]

For a more detailed view on the scenario and the semantics of the component types the reader is referred to [6]. For a precise description of the MECHATRONIC UML component type definition based on the UML Composite Structures and the differences to the previously used type modeling see [3].

3. COMPONENT STORY DIAGRAMS

Story Diagrams [2], a graph transformation formalism, offer a wide variety of features for the model-based specification of object-oriented behavior such as object/link creation/deletion, control flow structures and an appropriate visual representation. However, the structures conventional Story Diagrams operate on are simple objects connected by links. Components have ports connected among each other by assembly and delegation connectors. Secondly, Story Diagrams cannot traverse hierarchies of objects. In contrast to that, components explicitly are defined by composition and are arranged by other, more granular components and thus span a hierarchy. So the traditional Story Diagram approach does not align well with component-based architectures.

In [6], we introduced the new transformation language Component Story Diagrams, which combine the intuitive but formal notation of conventional Story Diagrams with the previously sketched, extended MECHATRONIC UML type definition. Instead of objects and links, component and port instances are created or destroyed and interconnected via assembly and delegation connector links. Calls to other Component Story Diagrams are supported to traverse the different hierarchy levels of a system configuration and to respect the encapsulation of components at the same time. The callee is defined either for the same component type or for a component type residing one hierarchy level below.

Figure 3 shows the Component Story Diagram initConvoySystem associated with ConvoySystem. This Component Story Diagram initializes a convoy system consisting of one component instance leader classified by convoyLeader (denoted by leader/convoyLeader) and several component instances /convoyFollowers (cf. Figure 2). The value of the parameter followers:Integer specifies the amount of /convoyFollowers instances to be created.

To achieve this behavior, firstly in Story Pattern createLeader the component instance leader/convoyLeader including two port instances is created by defining a corresponding component variable with ≪create≫ modifier and two attached port variables. Afterwards, the Component Story Diagram initConvoyLeader is called for the newly created component instance. This Component Story Diagram is defined for the component type RailCab residing one hierarchy level below and initializes the embedded configuration for leader. The subsequent statement activity creates a counter variable to create the required /convoyFollowers instances in a loop. The Story Pattern createNextFollower is executed as long as the loop condition holds. In this Story Pattern, a new component instance follower/convoyFollowers including three port instances is created and connected via the assembly connector :PosRef to a new port instance of the already bound leader. Whereas the call to initConvoyFollower initializes the embedded configuration of follower, the call to insertPosCalc changes the existing configuration of leader by deploying a new component instance /posCalc. To obtain the latter behavior, a name, the actual position, and the newly created port instance pos of leader are assigned as arguments.

The Fujaba4Eclipse editor for Component Story Diagrams is syntax-driven and context-sensitive w.r.t. the component type definition. While modeling a Component Story Dia-

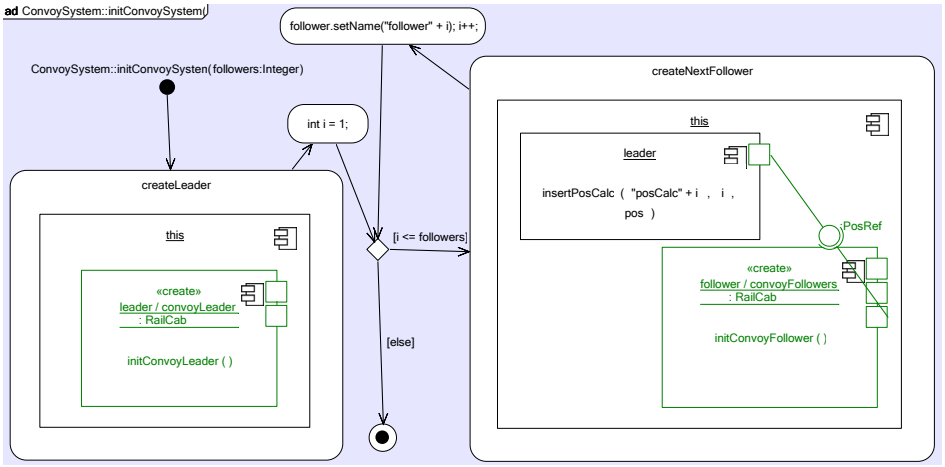


Figure 3: Component Story Diagram initConvoySystem [3]

gram for ConvoySystem, for example, as shown in Figure 3, the user can only assign parts to component variables which are defined in ConvoySystem, namely convoyLeader and convoyFollowers (cf. Figure 2). Both parts are interconnected by the assembly connector type PosRef attached to certain port types. Thus, you can only assign this connector type to the assembly link used in Story Pattern createNextFollower. Component variables can be associated with calls to Component Story Diagrams; the callee is selected from the Component Story Diagrams defined for the corresponding component type RailCab of the variable's part (cf. Table 1). Arguments representing a port variable, for example, have to be selected from the set of possible variables w.r.t. the signature of the callee. Further and in particular more complex Component Story Diagrams can be found in [3].

4. CODE GENERATION

To make use of existing code generation techniques, Component Story Diagrams are translated to conventional Story Diagrams based upon the MECHATRONIC UML metamodel. The translation is defined by TGG [5] rules, which thus describe the formal semantics of Component Story Diagrams. FUJABA's TGG plugins [7] are employed as concrete modeling and execution environment. A new code generation plugin takes the MECHATRONIC UML component type definition and Component Story Diagrams as input and transforms them using the translation rules to Story Diagrams classified by the metamodel.

Figure 4 shows a part of the Story Diagram which is generated from the Component Story Diagram of Figure 3.[2] The translation is for the most part straightforward. The control flow is basically unchanged. All variables in a Component Story Pattern are translated to objects of classes of the metamodel; for example, the component variable this is translated to the object parentComponentInstance:ComponentInstance. The variables' stereotypes are translated accordingly. Note the special case of the leader component variable translation. The translation honors the component type definition of Figure 2 which specifies that a convoy may only contain one leader RailCab. Adopting conventional Story Pattern semantics, an additional optional object leaderComponentInstanceToDelete is added to the generated Story Pattern which removes the old leader when a new one is created. Statement activities are taken over without modifications.

A more detailed view on the translation is sketched in [6], while the complete translation and an example of a translated Story Diagram is given in [3].

Executable code is generated from the translated Story Diagrams. Table 1 lists an overview of the used variables and links in Component Story Diagrams (second column) as well as in the translated Story Diagrams (third column) and of the generated Lines of Code (last column) for all evaluated Component Story Diagrams. This overview indicates the reduction of complexity by using Component Story Diagrams on MECHATRONIC UML component structures instead of modeling Story Diagrams based upon the MECHATRONIC UML metamodel. Note however that the generated LOC are not only related to the actual amounts of variables and links but to a greater extent to loops and iterations over $*$-associations.

The generated code can be executed at design time on component instance diagrams in Fujaba4Eclipse.

5. CONCLUSION AND FUTURE WORK

This paper introduced tool support for the modeling and the execution of sophisticated structural reconfigurations for component-based architectures by presenting an extended MECHATRONIC UML component type definition and a context-sensitive editor as well as a code generation plugin for Component Story Diagrams.

Currently there are two important open issues. First, Component Story Diagrams do not support negative or optional variables like conventional Story Diagrams. Since there are compositional dependencies between components and attached ports and between ports and linked connectors, the scope of a negative component variable, for example, includes also compositionally dependent variables. These have to be translated to a group of connected negative object vari-

[2]Actually, it is not a conventional Story Diagram but a slightly extended version which we chose due to implementation issues. The only difference in this example is the *transformation call* node. This node executes another Story Diagram after all specified changes like creating and destroying edges and nodes have been executed.

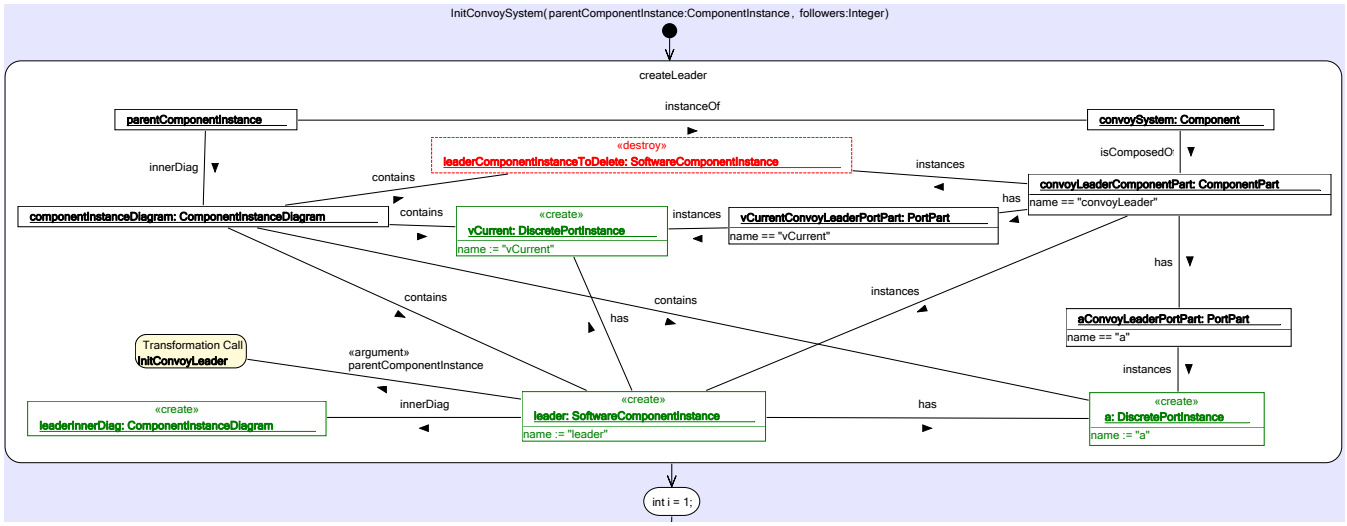


Figure 4: Excerpt of Story Diagram generated from Component Story Diagram initConvoySystem

Comp. Story Diag.	Variables and connector links	Object variables and links	LOC
ConvoySystem::initConvoySystem	12	89	760
ConvoySystem::insertFollower	8	69	605
ConvoySystem::removeFollower	9	71	618
RailCab::initConvoyFollower	11	89	695
RailCab::initConvoyLeader	15	137	968
RailCab::insertPosCalc	52	482	2806
RailCab::removePosCalc	42	389	2422
Overall	**149**	**1326**	**8874**

Table 1: Comparison of amounts of variables/links in Component Story Diagrams and in translated Story Diagrams and of generated Lines of Code [3]

ables in the resulting translated Story Diagram. Until now, conventional Story Diagrams forbid such negative variable groups, what also affects groups of optional variables. A concept has already been developed to solve this problem [3] but has not been implemented yet. Secondly, the presented Component Story Diagrams are selected and executed at design time by an user. To obtain a self-adaptive approach for mechatronic real-time systems, we are currently working on an integration of Component Story Diagrams into real-time statecharts.

Future works could include extensions like attributes as well as an inheritance concept for component types to provide Component Story Diagrams with attribute conditions and the matching of instances of derived component types.

6. REFERENCES

[1] S. Burmester, H. Giese, and M. Tichy. Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In U. Assmann, A. Rensink, and M. Aksit, editors, *Model Driven Architecture: Foundations and Applications*, volume 3599 of *Lecture Notes in Computer Science*, pages 47–61. Springer Verlag, Aug. 2005.

[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* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.

[3] J. Holtmann. Graphtransformationen für komponentenbasierte Softwarearchitekturen. Master's thesis, University of Paderborn, Germany, April 2008.

[4] Object Management Group. *UML 2.0 Superstructure Specification*, October 2004. Document: ptc/04-10-02 (convenience document).

[5] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proc. of the* 20^{th} *International Workshop on Graph-Theoretic Concepts in Computer Science*, Herrschin, Germany, June 1994. Spinger Verlag.

[6] M. Tichy, S. Henkler, J. Holtmann, and S. Oberthür. Component Story Diagrams: 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*, 2008.

[7] R. Wagner. Developing Model Transformations with Fujaba. In H. Giese and B. Westfechtel, editors, *Proc. of the* 4^{th} *International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275 of *Technical Report*, pages 79–82. University of Paderborn, September 2006.

Towards Software Product Line Testing using Story Driven Modeling

Sebastian Oster, Andy Schürr, Ingo Weisemöller
Real-Time Systems Lab, TU Darmstadt
{oster, schuerr, weisemoeller}@es.tu-darmstadt.de

ABSTRACT

This paper provides an approach towards model based Software Product Line Testing using Story Driven Modeling. The motivation is to identify a small set of Product Instances such that the verification of this test set guarantees the correctness of all possible Product Instances or even the whole Product Line. One of the most famous approaches to model variability in Software Product Lines is the feature model. The feature model is constructed based on functionality and system requirements. To simplify the feature management and testing we realize the test set selection on the basis of the feature model. Dependencies derived from the system architecture are also added. We use Fujaba/MOFLON to specify the feature model and to describe methods operating on this model. Therefore, each feature selection and any algorithm used for testing can be formulated using graph transformation with Story Driven Modeling (SDM). This paper is a first discussion of our approach.

Keywords

Software Product Line, Combinatorial Testing , Story Driven Modeling, Feature Model

1. INTRODUCTION

Software Product Lines are an approach to improve reusability of software components in a large number of products that share a common set of features. In this paper we focus on decreasing the test effort for Software Product Lines. A Software Product Line can provide an enormous number of different Product Instances. Testing each Product Instance individually has proven itself as a trustworthy but extensive method. To decrease this effort we use the combinatorial test design and adapt it to our approach. The idea of using a combinatorial test design is not a new but a very promising approach. With the best of our knowledge combinatorial testing is mainly used to reduce the amount of test parameters [1]. Only certain combinations of test parameters are used for testing instead of testing each test parameter individually or all possible combinations. In this paper we apply combinatorial testing to generate a set of Product Instances. It is a mathematical method to select certain, adequate samples of combinations of Product Instances. Testing this set of Product Instances we assume that it results in a verification of all possible Product Instances. In our approach we also use the feature model to represent dependencies between features and groups of features of a Software Product Line. Regarding these dependencies, certain features are chosen using combinatorial testing. These dependencies influence the selection of certain Product Instances that implement the selected combinations of features. A feature model represents the set of features hierarchically and captures commonality and variability. We use the notation according to the BMBF feasiPLe Project [3]. Feature models in feasiPLe consist of optional, mandatory, xor, and alternative features. A feature can be a variation point or a variant. Each feature which splits into other features is called a variation point. A variant is simply a feature which cannot be subdivided into other variants. We will explain the notation and functionality roughly with the assistance of our running example a Software Product Line for LEGO NXT model cars.

2. RELATED WORK

Several papers cover the subject Product Line testing. However these work mainly focuses on the question how to integrate testing in the overall Product Line development process [6] or, how to generate and organize test cases [4]. In [5] the author also tries to minimize the amount of Product Instances which need to be tested to verify the whole Software Product Line. However the work is based on a different representation of variability and the calculation of the minimal test set is intricate and very time consuming. We refer to [5] for further details.

3. RUNNING EXAMPLE

We present a Software Product Line of LEGO model cars. We use a feature model to model variability. It is a very simplified and abstract example in comparison to real vehicles. The functionality is based on Lego Mindstorm NXT actors, and sensors. The corresponding feature model (cf. Fig.1) covers only a few features of the model car we use to evaluate our ideas in Software Product Line testing, but should suffice to present the basic ideas of our approach. For the following explanation please consult the corresponding feature model. The feature model consists of software features (driving/steering mode & driver support) and hardware features (sensors & actuators). Driving mode offers three different types of locomotion. These are autonomous driving ("Automotive"), random drive and steering movements ("Crazy"), and "Manual" driving which requires user interacion. All features are alternatives but at least one of these features must occur in a Product Instance. If "Manual" driving mode is selected one has to choose between gamepad or steering wheel control. These features exclude each other so only one of it can be included in a product instance. We name this notation xor for exclusive-or. The variation point driver support offers different optional software features to support driv-

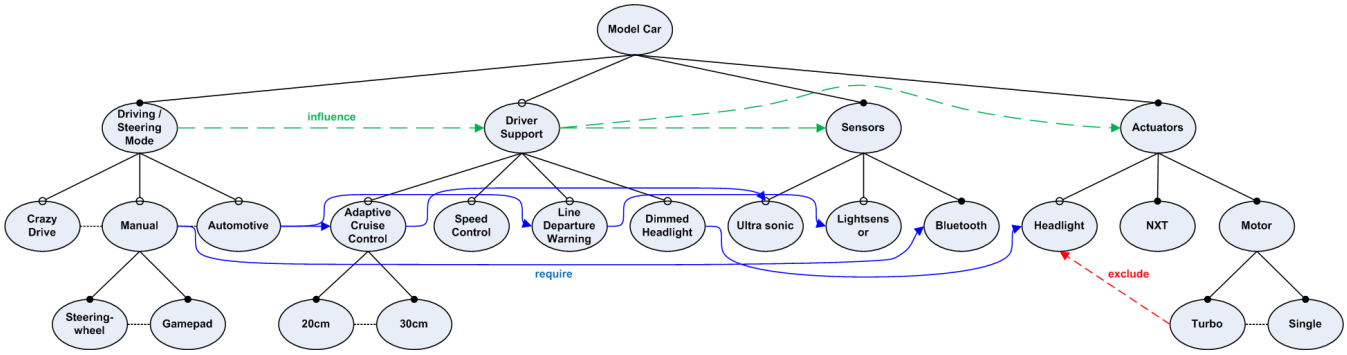


Figure 1: LEGO Model Car Product Line

ing. These are Adaptive Cruise Control (”ACC”), Lane Departure Warning (”LDW”), Dimmed Headlight, and Speed Control. All features are optional which means that Product Instances can be derived that do not include any of these features. If ”ACC” is chosen for a certain product, one has to decide what degree of sensitiveness is required. The last two variation points ”sensor” and ”actuator” are the hardware components. Both come along with Lego Mindstorm NXT Set. We refer to [2] for more detailed information. The variation point ”actuator” includes mandatory features (”motor” & ”nxt”). That means that these features have to be included in each derived Product Instance.

4. PRODUCT INSTANCE SELECTION

Instead of testing all possible Product Instances individually we use combinatorial testing to identify a minimal test set of Product Instances. Our strategy is based on two major concepts to reduce the test effort for Software Product Lines. Exploiting dependencies between features and combinatorial testing. The basic idea is that we first of all have somehow to identify dependencies (independencies) between features or groups of features relying e.g. on user input, static analysis of code or architectures or inspection of runtime traces of the regarded Software Product Line. The identificaton of these dependencies is out-of-scope of this paper. Afterwards a minimal set of Product Line instances has to be determined that covers all possible combinations of pairs of features. The computation of such a test set is an NP-complete problem [5]. Therefore, we have to combine strategies from greedy and constraint-solving algorithms to compute reasonably small test sets of Product Instances effectively. We have just started first experiments to play with different combinations of strategies which cannot be presented here due to lack of space. As a consequence we will only present the basic ideas of our approach and give the reader an impression why we have selected SDM as an ideal algorithm prototyping language for this purpose. Generating a set of Product Instances for testing is a multilevel process. Before we go into any detail we sketch our approach to give an overview over the different levels of our procedure.

1. Analysing Dependencies
   The feature model serves as basis for our approach. First of all we need to add dependencies between the different features to our feature model. These dependencies can be derived e.g. from system specifications, hardware restrictions or system requirements. E.g. in our running example the model car can only provide the feature Adaptive Cruise Control (ACC) if an ultrasonic sensor is installed. Also, if two motors are used to accelerate the vehicle (turbo) we have no plug-in position left for the light. Therefore, the features light and turbo motor exclude each other. Additionally we need to add dependencies derived from the software architecture. In contrast to the feature dependencies we try to apply these dependencies between the variation points. If we can not pass on the dependencies from the architecture to the variation points directly we can consider a reorganisation or adaption of the existing variation points. But this is not the focus of our approach. All in all we want to have a feature model which is representative for the functionality and the architecture dependencies. To simplify our running example we use the same variation points directly beneath the root node as we have software and hardware components in our architecture. To clarify the difference between the two kinds of dependencies we name them differently. Require and exclude for dependencies between features and influence edges for links between variation points.

2. Normalizing the feature model
   Since we want to realize the combinatorial test design using graph transformations appropriate transformation rules have to be specified. We do not want to test all possible feature combinations but a minimal set of Product Instances. To simplify the selection for combinatorial testing we normalize the feature model. The normalization ”flattens” the feature model so that only three levels remain. These are the root-feature level (model car), the variation point level including dependencies derived from the architecture, and the variant level including dependencies between variants. Instead of using mandatory, optional and alternative variants we realize graph transformation rules which replace these features using only xor variants to represent the same functionality. For lack of space we do not list SDM rules for this graph transformation. We can also unify the dependencies between the features. Instead of saying that a variant requires a certain other variant we can say that it excludes all other variants. We illustrate the normalization in the appendix. To ensure a clear overview we only drew one exclude de-

pendency (cf. Fig.3).

3. Combinatorial Testing
   In this step we assemble Product Instances based on pairs of variants. We only want to generate Product Instances of pairs of variants which depend on each other. The variation points which are linked by an dependency edge depend on each other and so do all variants beneath them. Therefore, we only have to combine pairs between dependent variation points. According to the normalized feature model of our running example we have to pair each variant of "Driving/Steering Mode" with each variant of "Driver Support". We also have to pair "Driver Support" - "Sensors" and "Driver Support" - "Actuators". However, we have to consider the exclude edges between the features. For Instance, if we want to pair the variant combination including "Automotive" with a variant combination of "Driver Support" we have to ensure that the variant combination does not exclude "ACC" and "LDW". This decreases the amount of possible Product Instances. We describe the generation of Product Instances in detail in the next section.

Please consult the feature models of our running example for details for the first (cf. Fig.1) and the second step of our procedure (cf. Fig.3). We marked the two different types of dependencies. The require (marked blue) and exclude (marked red) dependencies between features and additional dependencies derived from the software architecture (marked green). The dependencies on feature level are used to ensure that only reasonable feature combinations are composed during the combinatorial test. Edges between variation points give information about which variants of each variation point interdepend. We will only pair variants of variation points which interdepend.

5. GRAPH TRANSFORMATION

Considering the lack of space of this paper we can only describe one of the graph transformation steps with SDM. Since we see a higher benefit in presenting the combinatorial testing, we use SDM to formulate transformation rules for the combinatorial test design. However we want to give a short insight into the normalization procedure of the feature model.

5.1 Peek insight feature model normalization

Flattening the feature model is a bottom-up process. We start at the lowest level and pass up the variants to the variation point above. We describe this procedure giving an example. We assume that each variant has its name as an attribute. Converting two alternative variants $v1$ and $v2$ into a xor related feature we first need to create a new variant which name is a concatenation of the source variants. We therefore need a rule to specify that we can only concatenate two names if `variant1.name<variant2.name`. Else we would get two combinations $v1v2$ and $v2v1$. This new variant becomes an xor feature. In the next step the alternative edges from the variation point to the variants are deleted and replaced by xor edges. The result of two alternative variants are three xor variants $v1$,$v2$, and $v1v2$. Analog to this procedure we can also convert optional and mandatory features to xor features. A realization with SDM is possible. The normalization of our running example is illustrated in the appendix.

5.2 Combinatorial Selection using SDM

Our combinatorial testing approach generates certain Product Instances which include pairs of variants. Generally, we pair each variant of a variation point $VP1$ with each variant of variation point $VP2$ if $VP1$ and $VP2$ are connected by an influence edge. We have to consider the dependencies between variants and pair only variants which do not exclude each other. These pairs of variants need to be combined to form Product Instances. We use the dependencies derived from the architecture to pair interdepending variants and special rules to form the Product Instances. To realize this approach we iterate over the influence edges on the variation point level. We implement a container class which realizes a table of Product Instances. We use SDM to model our approach (cf. Fig.2). We enumerated the different activities in the SDM diagram.

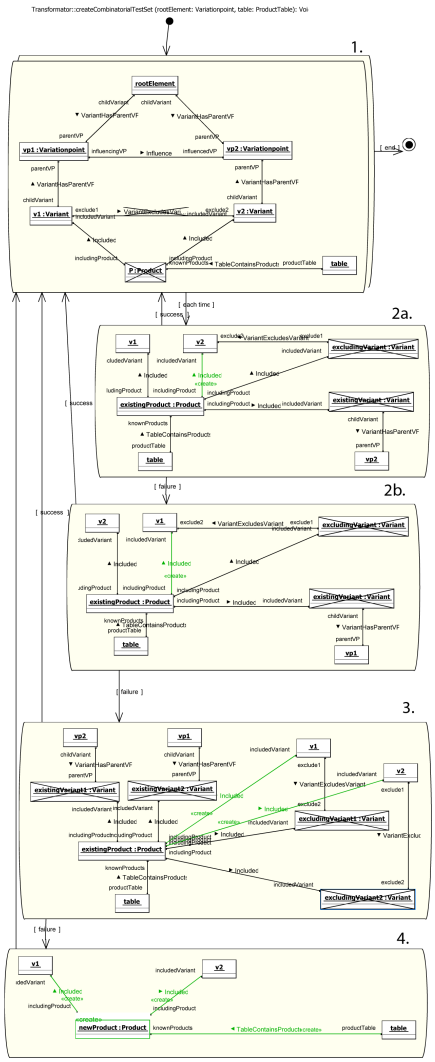


Figure 2: SDM Diagram for generating Product Instances

In the first activity, all pairs of variants which influence each other are determined. Variants influence each other if they 1. do not exclude each other and 2. belong to two variation points which are linked by a dependency edge. If this pair of variants is not covered by an existing product instance we apply the following rules to try to extend an existing product instance.

1. We check if an existing Product Instance already covers one of those variants and if the variation point of the other variant is still unbound. We can then use activity 2a or 2b to add the mentioned variant to an existing Product Instance.
2. If the rule above fails the third activity will look for a product instance in which the two concerning variation points of the variants are still unbound.

If no Product Instance can be found which can be extended we have to create a new Product Instance.

6. CONCLUSION

We can use SDM to select certain Product Instances for testing and for the normalization of the feature model. This is our first approach using graph transformation on feature models to realize combinatorial testing for Software Product Lines. We have to examine different and very complex algorithms which can operate on graphical structurs. Fujaba/MOFLON is an ideal tool for high-level specification and the rapid prototyping of these algorithms. Since the calculation of a minimal test set is a NP-complete optimization problem we can not guarantee that our test set of Product Instances is minimal. However we decreased the amount of Product Instances which need to be tested dramatically in comparison to testing all Product Instances.

7. REFERENCES

[1] D. M. Cohen, S. R. Dalal, A. Kajla, and G. Patton. The automatic efficient tests generator. *Fifth Int'l Symposium on Software Reliability Engineering*, IEEE:303–309, 1994.

[2] LEGO. www.lego.com.

[3] F. P. Page. www.feasiple.de.

[4] K. Pohl and A. Metzger. Software product line testing. *Communications of ACM*, 49(12), 2006.

[5] K. Scheidemann. Verifying families of system configurations. *Doctoral Thesis*, TU Munich, 2007.

[6] A. Tevanlinna, J. Taina, and R. Kauppinen. Product family testing: a survey. *ACM SIGSOFT Software Engineering Notes.*, 29, 2004.

APPENDIX

A. NORMAILZED FEATURE MODEL

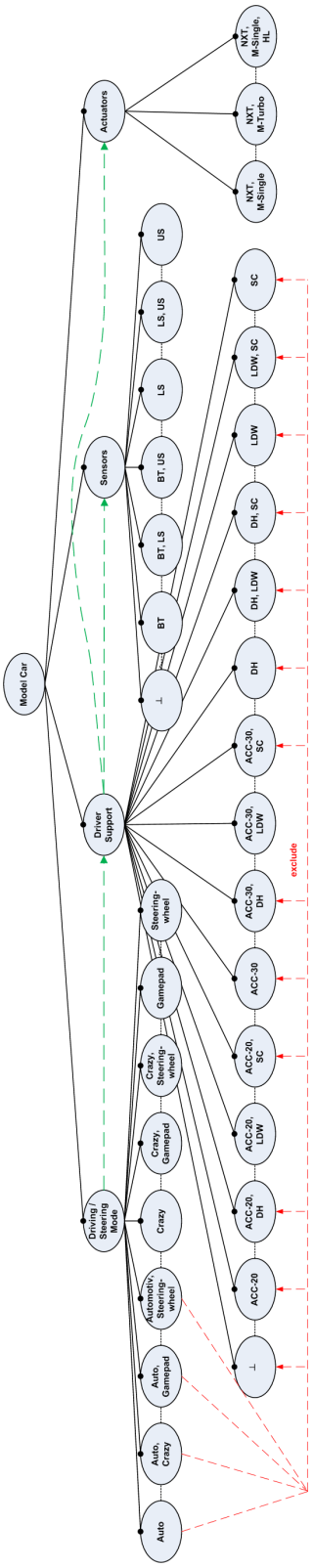


Figure 3: Normalized Feature Model

Integration of Legacy Components in MechatronicUML Architectures*

Christian Brenner, Stefan Henkler, Martin Hirsch, and Claudia Priesterjahn
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[cbr|shenkler|mahirsch|cpr]@uni-paderborn.de

Holger Giese
System Analysis and Modeling Group
Hasso Plattner Insitute
University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
D-14482 Potsdam, Germany
holger.giese@hpi.uni-potsdam.de

ABSTRACT

One of the main benefits of the component-based development paradigm is its support for reuse which is guided by the interface description of the components. This facilitates the construction of complex functionality by the flexible composition of components. However, the also required verification of the resulting system often becomes intractable in practice as no abstract model of the reused components, which can serve the verification purpose, is available for the integrated legacy components. In this paper, we present the integration of legacy components in a MECHATRONIC UML model by an incremental synthesis of the communication behavior of the embedded legacy components combined with compositional verification.

1. INTRODUCTION

One of the main benefits of the component-based development paradigm is its support for reuse which is guided by the interface description of the components (cf. [10, 2]). In general, the proper composition of independent developed components in the software architecture of embedded real-time systems requires means for a sufficient verification of the integration step either by testing or formal verification. However, the overwhelming complexity of the interaction of distributed real-time components usually excludes that testing alone can provide the required coverage when integrating legacy components.

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 which can serve the verification purpose is available for legacy components.

A number of techniques which either use a black-box approach and automata learning [8] or a white-box approach which extracts the models from the code [9, 1, 7] exists. However, these approaches did not consider the specific context for efficiently synthesizing the relevant behavior of the legacy component, which is of paramount importance for embedded systems. Further, these approaches are not capable of finding conflicts in early learning steps.

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

In this paper we present a tool support for the incremental synthesis of communication behavior for embedded legacy components by combining compositional verification and model-based testing techniques based on [4, 6]. For the exploration of the component's behavior a formal model of the component's environment is applied. The environment model is employed to derive known environment behavior which is then used to systematically synthesize the relevant behavior of the legacy component as well as a formal model describing its communication behavior. While this formal model is not a valid encoding of all possible behavior of the legacy component, it is in fact a valid representation of its communication behavior for the context relevant for its embedding.

In the next section we present the incremental synthesis approach. Afterwards we describe the implementation as well as the evaluation. We finish the paper with the conclusion and future work.

2. INTEGRATION OF LEGACY COMPONENTS

Given a concrete context and a concrete component implementation with hidden internal details (legacy component), the basic question we want to check is whether a given property ϕ as well as deadlock freedom ($\neg\delta$) holds. We are in particular interested in a guarantee that both properties hold or a counterexample witnessing that they do not hold. However, usually the legacy component cannot be employed to traverse the whole state space as the state space of the complete system is too large to directly address this question. Before we answer the question, we discuss in the next section the prerequisites our approach. Afterward, we present an overview of our approach and discuss in more detail the relevant technique in Section 2.3.

2.1 Prerequisites

The approach presented in this paper will only work, if certain prerequisites and constraints can be fulfilled by the legacy component. The component must have neither nondeterministic nor pseudo-nondeterministic behavior. For example the firing of transitions must not directly depend on variable values or timing constraints since they are currently not explicitly captured. Such conditions will only be valid,

if they are encapsulated in the state information. All transitions must fire within a given timespan after the receipt of a triggering message. That is to prevent that a test run which leads to a deadlock will not terminate. This timespan also applies to ϵ-transitions. In this case the timespan starts at the entry to the transition's start state. To enable the learning of the behavior the definitions of the component's interfaces muss be known, a start state must be given and it must be possible to reset to this start state. Further, the state changes (current state) must be observable at the interface.

Our experiences showed us that the prerequisites, besides the state information, are realistic for mechatronic systems, as this are reactive systems. As discussed in the future work we have to extend our black-box approach with additional white-box information, to abandon on the state information at the interface of the legacy component.

2.2 Sketch of the Proposed Approach

Given a MECHATRONIC UML architecture which embeds a legacy component and behavioral models for all other components building the context of the legacy component, the basic question of correct legacy component integration is whether for the composition of the legacy component and its context all anomalies such as deadlocks are excluded or all additionally required properties hold. However, it is usually very expensive and risky to reverse-engineer an abstract model of the legacy component to verify whether the integration will work.

To overcome this problem we suggest employing some learning strategy via testing to derive a series of more detailed abstract models for the legacy component. The specific feature of our approach will be that we exploit the present abstract model of the context to only test relevant parts of the legacy component behavior. The approach depends only to a minimal extent on reverse engineering results.

We start with synthesizing a model of the legacy component behavior based on known structural interface description. As shown in [4] we use a safe over approximation. Then, we check whether the context plus the model of legacy behavior exhibit any undesired behavior taking generic correctness criteria or additional required properties into account. If not, we use the resulting counterexample trace to test the legacy component. If the trace can be realized with the legacy component, a real error has been found. If not, we first enrich the trace with additional information using deterministic replay [3] and then merge the enriched trace into the model of the legacy component behavior. We repeat the checks until either a real error has been found or all relevant cases have been covered.

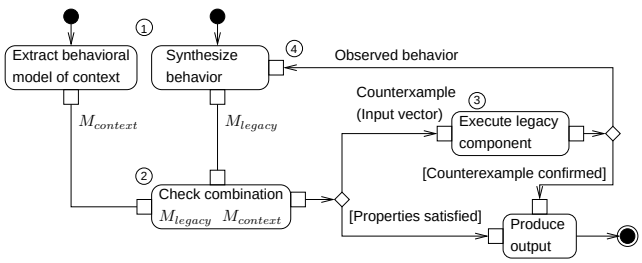


Figure 1: Sketch of the approach

Figure 1 illustrates our process with a summary of the overall approach. 1) Initially, we synthesize an initial behavior model for the legacy component based on known structural interface description and derive a behavioral model of the context from the existing MECHATRONIC UML models. 2) We check the combination of the two behavioral models and either get a) a counterexample or b) the checked properties are guaranteed. In the latter case we are done. 3) If we have a counterexample, we use this as test input for the legacy component. Deterministic replay enables us to enrich the observable behavior with state information by monitoring. If the tested faulty run is confirmed, we have found a real counterexample. If not, we can use the new observed behavior to refine the previously employed behavior model of the legacy component. We repeat steps 2) to 4) until one of the described exits occurs.

The approach can be extended to multiple legacy components, by using the parallel combination of multiple behavioral models. The iterative synthesis will then improve all these models in parallel. While theoretically possible, we can currently provide no experience whether such a parallel learning is beneficial and useful for multiple legacy components. Our expectation that it depends on the degree in which the known context restricts their interaction which determines which benefits our approach may show also for this more advanced integration problems.

2.3 Chaotic Closure

For our approach it is necessary that the model checker takes into account every behavior which is possible according to our current knowledge about the system. To accomplish this, the already known parts of the system are extended with chaotic behavior, resulting in a new model called *chaotic closure*. The latter is then, in combination with a model of the context, subject to model checking. Namely, for all so far unknown behavior it is assumed that on the one hand *any* possible interaction may occur but on the other hand a deadlock is possible at any time as well. Therefore, the chaotic closure is an over approximation of the real system: It always models at least all of the system's behavior, but not all of the modeled behavior has to be possible in the system.

For modeling chaotic behavior a *chaotic automaton*, a non-deterministic finite automaton consisting of two states, can be used: The state s_δ with no outgoing transitions represents the case of the system being in a deadlock, neither receiving nor sending any messages. The state $s_\forall$ on the contrary represents the case where all inputs being possible for the system are enabled and all outputs can occur. This is modeled by one self-transition and one transition to s_δ for each possible input (with no output) and each possible output (with no input). For creating these transitions the input- and output-alphabets of the system must be known. Both states of the chaotic automaton are initial states.

The chaotic closure is a combination of the synthesized model with the chaotic automaton for the system, mapping all unknown behavior to a chaotic one. Figure 2 shows as an example the Chaotic Closures (on the right side) for a trivial first conjectured behavior model and for a slightly more advanced one.

A Chaotic Closure is constructed as follows: First, the chaotic automaton for the input- and output-alphabets of

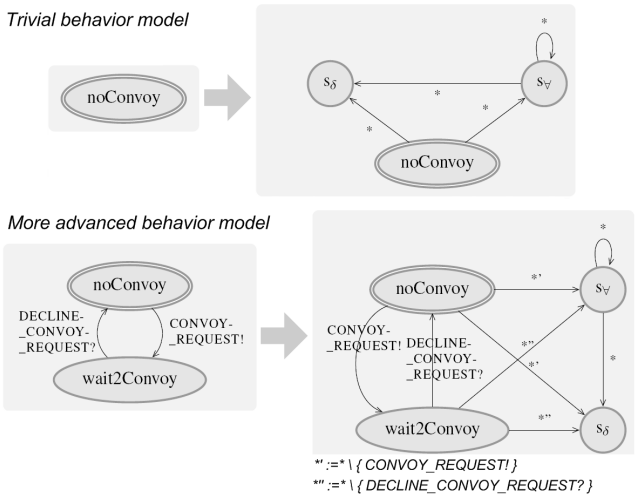


Figure 2: Example for a chaotic closure

the system is constructed. Then the states and transitions of the chaotic automaton are added to the incomplete automaton modeling the behavior that has been learned until now. For every combination of a state and an incoming or outgoing event for which a transition neither has been defined nor excluded, a new transition is created from that state to both the $s_\forall$ and the s_δ state. Contrary to the synthesized behavior, the chaotic closure constructed for it is non-deterministic.

The explicit deadlock state s_δ in the chaotic hull makes sure that as long as there still is behavior left to learn, the model checker will be able to find a deadlock. The result is that in every iteration of our approach at least one new transition is learned. However this only applies to behavior of the system which can be reached in combination with the model of the context. Any other behavior is not considered to be relevant in the context the system is integrated into, and therefore no time needs to be wasted with testing it.

3. IMPLEMENTATION AND EVALUATION

As the aim of our synthesis approach is to allow to safely integrate legacy components into existing Mechatronic UML contexts, it has been implemented in a way that it is on the one hand compatible to models created with the Fujaba Real-Time Tool Suite and on the other hand does not rely on a specific testing framework. The latter is important because it depends on the legacy component, which testing framework can be used. To accomplish this, the behavior synthesis step, being the core of the approach, was implemented in Java as a command-line tool. For the verification step, the model checker verifyta of the integrated tool environment UPPAAL has been chosen.

In the synthesis tool automata are saved as *Extended Hierarchical Timed Automata* (ExHTA), which have the same semantics as Fujaba Real-Time Statecharts (RTSCs), in a tool-independent XML-format. This has several advantages: As RTSCs can be exported into this format, it is possible to use a context which has been modeled in Fujaba using Mechatronic UML. Also the Uppaal Plugin provides a way to convert ExHTA to Timed Automata which can be validated using verifyta. It is possible to enable support for other model checkers as well, by implementing additional mappings to the formats they are using. Finally, the model synthesized by our approach also is saved in ExHTA. This should simplify implementing a method for loading it as a RTSC in Fujaba[1].

The first execution of the synthesis tool creates the first trivial behavior model, it constructs the corresponding chaotic closure by using the system's i/o-interface and it combines it with the model of the context. The subsequent execution of the (slightly modified) Uppaal Plugin converts this combination from ExHTA to the Uppaal XML-format. Then verifyta is used for model checking it against the properties defined in a certain CTL dialect. The resulting counterexample (if any) is then used on the one hand by a testing framework to execute it as a test case, on the other hand it is used by the synthesis tool for comparing it against the trace resulting from those tests. If trace and counterexample conform, a message will be issued by the tool. Otherwise the trace is used for learning new behavior (and so on).

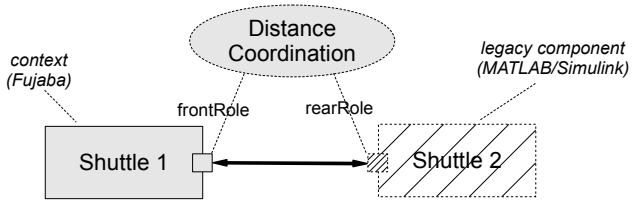


Figure 3: Component diagram for the scenario used for evaluation of our approach

Using the synthesis tool, our approach has been evaluated within a MATLAB simulation for the case of a MATLAB/Simulink legacy component being integrated into an existing Mechatronic UML system model. Figure 3 shows a component diagram picturing the scenario considered for this. The components in this diagram are two Rail Cab Shuttles, safety critical mechatronic systems, communicating with each other within a collaboration pattern. One of these Shuttles, the one that made the decisions, was the context in our approach. The other one, which was a reactive system, was the system to be integrated.

To realize the evaluation, a simple testing framework was implemented within MATLAB/Simulink. The main task of this framework is the execution of the counterexamples provided by the model checker as test cases: It sends a message to the system whenever the context would send one according to the counterexample and it logs all communication and all state changes of the system. Additionally, by calling the synthesis tool and the model checker it is able to execute our approach automatically for this scenario.

The evaluation showed that our approach is able to successfully synthesize a correct model for the given scenario. Also a simple error that had been added to the simulation could be found. However, it turned out that the synthesis needed quite a few steps, especially for learning the complete correct model. This was due to the model checker returning only very short counterexamples which lead to only one transition being learned in each iteration of the approach.

[1]An implementation for this is currently under development.

The total amount of testing necessary was greatly increased by this because to reach a new transition, often much of the already known behavior had to be tested again.

A way to force the model checker to create longer counterexamples is instrumenting the Chaotic Closure and modifying the temporal logic formula used by the model checker. These changes can be used together with certain command line options to make verifyta try to maximize the number of some transitions in the counterexamples. Three different modifications of this kind have been tried: One possibility is to increase the number of self-transitions of $s_\forall$ up to a maximum value. However this has the drawback that verifyta usually uses several iterations of one loop to achieve this. Another option is to make the model checker try to use every self-transition of $s_\forall$ at least once in each counterexample. Finally, the model checker can also be driven to try to let every counterexample contain every transition of the context's model. Each of these possibilities has been tested for our evaluation scenario, in several cases resulting in a much smaller amount of iterations and a smaller total amount of testing steps as well. However the success of each of these modifications is likely to depend heavily on the specific scenario they are used in.

In addition to the evaluation with MATLAB/Simulink we also consider using our approach with the IPANEMA framework. Using our synthesis approach within this framework would have the advantage of being able to use the model of the context on the one hand for exporting it to ExHTA and on the other hand for automatically generating code from it which can be compiled to run on the framework. A testing framework for model based testing [5] already exists which can be adapted to work in conjunction with the synthesis tool. Also a Test Case Generator exists which can be used for converting counterexamples to test cases.

4. CONCLUSION AND FUTURE WORK

In this paper we have presented a tool support for the incremental synthesis of communication behavior for embedded legacy components by combining compositional verification techniques and model based testing. It enables context specific learning with conflict detection in early learning steps. The employed learning strategy provides options for optimization as shown in the evaluation. The interplay between the formal verification and the test could be improved when a number of counterexamples instead only single one could be derived from the model checker. This is achieved by using specific strategies of the model checker to derive counterexamples.

Next, we want to combine the presented dynamic analysis with a static analysis to extend the applicability of the approach. E.g., the parts of the code which are responsible for the current state or internal variable values and dependencies could be detected by a static analysis and used by the dynamic analysis.

5. REFERENCES

[1] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[2] I. Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[3] H. Giese and S. 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.

[4] H. Giese, S. Henkler, and M. Hirsch. Combining Compositional Formal Verification and Testing for Correct Legacy Component Integration in Mechatronic UML. In R. de Lemos, F. D. Giandomenico, C. Gacek, H. Muccini, and M. Vieira, editors, *Architecting Dependable Systems V*, volume 5135 of *LNCS*, pages 248–272. SPRINGER, 2008.

[5] H. Giese, S. Henkler, M. Hirsch, and C. Priesterjahn. Model-based testing of mechatronic systems. In L. Geiger, H. Giese, and A. Zündorf, editors, *Proc. of the 5th International Fujaba Days 2007, Kassel, Germany*, pages 1–4, September 2007.

[6] S. Henkler and M. Hirsch. Compositional validation of distributed real time systems. In *OMER4 - Object-oriented Modeling of Embedded Real-Time Systems*, pages 1–6, 2007. accepted.

[7] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 597–607, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[8] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *In Proc. 15 Int. Conf. on Computer Aided Verification*, 2003.

[9] D. Lucio, J. Kramer, and S. Uchitel. Model extraction based on context information. In *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems*, LNCS. Springer, 2006.

[10] C. Szyperski. Component Software and the Way Ahead. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, incollection 1, pages 1–20. Cambridge University Press, New York, NY, 2000.