Holger Giese, Albert Zündorf (Eds.)

13.-14. Oktober 2003, Kassel, Germany

Proceedings

Holger Giese, Albert Zündorf (Ed.)

Fujaba Days 2003

13.-14. Oktober 2003, Kassel, Germany

Volume Editors

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

Dr. Holger Giese
University of Paderborn, Institute for Mathematics, EIM
Warburger Straße 100, 33098 Paderborn, Germany
hg@uni-paderborn.de

Program Committee

Albert Zündorf
University of Kassel, Germany

Tarja Systä
Tampere University of Technology, Finland

Wilhelm Schäfer
University of Paderborn, Germany

Luuk Groenewegen
Leiden University, Netherlands

Holger Giese
University of Paderborn, Germany

Local Organisation

Albert Zündorf, Rose-Marie Biehlig, University of Kassel

Editors' preface

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

At the early days, Fujaba had a special focus on code generation from UML diagrams resulting in a visual programming language. Today, at least four rather independent tool versions are under development in Paderborn and Kassel for supporting (1) reengineering, (2) embedded systems, (3) the Fujaba Development Process, and (4) education. According to our knowledge, quite a number of research groups have also chosen Fujaba as a platform for their own UML related research. In addition, quite a number of Fujaba users send us requests for more functionality and extensions.

Therefore, the 1st International Fujaba Days aim at bringing together Fujaba developers and Fujaba users from all over the world to present their ideas and projects and to discuss them with each other and with the Fujaba core development team.

Actually, we have managed to attract contributions from Finnland, Belgium, England and from three different sites in Germany. In addition, there are other groups in Canada, the Netherlands, Sweden and Italy that work on and with Fujaba but did not manage to submit for this year's workshop.

To provide maximal benefits and to give anybody a chance to meet and to bond to all the other peoples working in the Fujaba context, in addition to the talks and tutorials, we have reserved a lot of time for discussions and working groups.

We hope, that you enjoy the workshop.

The editors

Table of Contents

Sven Burmester, Holger Giese
The Fujaba RealTime Statechart PlugIn 1

Martin Hirsch, Holger Giese
Towards the Incremental Model Checking of Complex RealTime UML Models 9

Matthias Tichy, Margarete Kudak
Visualization of the execution of Real-Time Statecharts 13

Pieter Van Gorp, Niels Van Eetvelde, Dirk Janssens
Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Metamodel 17

Leif Geiger, Christian Schneider, Albert Zündorf
Integrated, Document Centered Modelling in Fujaba 25

Carsten Amelunxen, Alexander Königs, Tobias Rötschke, Andy Schürr
Adapting FUJABA for Building a Meta Modelling Framework 29

Kalle Aaltonen, Jyrki Nummenmaa, Timo Poranen
Layout Algorithms for FUJABA Diagrams 35

Ari Seppi, Jyrki Nummenmaa
A Database Schema Diagram Plugin for Fujaba 39

YC 'Vik' Nuckchady
Turning FUJABA into a Collaborative Tool 45

Tutorials

Lothar Wendehals
10 Steps to build a Fujaba Plug-In 47

Matthias Tichy
How to add a new diagram to Fujaba 55

Susannah Moat
Adapting the Fujaba Code Generation Mechanism 75

Ira Diethelm, Leif Geiger, Albert Zündorf
Story Driven Modeling and programming with Fujaba 99

The Fujaba Real-Time Statechart PlugIn

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

ABSTRACT

Distributed embedded real-time systems are one of the most successful application areas of the UML. However, the UML techniques for behavior modeling such as Statecharts in their current form do not support real-time as required, because of the unrealistic underlying zero-time execution assumption for side-effects. With Real-Time Statecharts, a related extension has been developed for the Fujaba Tool Suite that overcomes these limitations by supporting a well-defined real-time semantics based on Timed Automata and code synthesis which guarantees the specified timing characteristics. Besides the Real-Time Statecharts the paper describes the currently available tool support and the underlying principles of the code generation for the currently supported platform, Real-time Java.

Keywords

Statecharts, Real-Time, Embedded Systems, UML, Fujaba.

1. INTRODUCTION

Today, the *Unified Modelling Language (UML)* [13] is successfully applied to model complex embedded systems. However, the standard UML behavior modeling techniques such as Statecharts are not appropriate in their current form. For distributed real-time systems, the underlying zero-time execution assumption for side-effects is often unrealistic and conflicts with a consistent implementation of the high level UML model on available hardware and software platforms.

With Real-Time Statecharts [4, 6], a related extension has been developed for the Fujaba Tool Suite that overcomes these limitations by supporting a well-defined real-time semantics based on Timed Automata and code synthesis guaranteeing the specified timing characteristics.

The tool support currently available for Real-Time Statecharts in form of a Fujaba PlugIn consists of an extended diagram notation, a time consistency checker, and code synthesis. Additionally, a function permits to check whether multiple Real-Time Statecharts can be scheduled on a single node using the timing information extracted from the real-time UML model.

*Supported by the International Graduate School of Dynamic Intelligent Systems. University of Paderborn

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

In Section 2 we will first review the requirements for real-time modeling with UML and the shortcomings of the current UML support. Then, Real-Time Statecharts are described informally in Section 3. Section 4 guides how to use the Real-Time Statecharts PlugIn for the Fujaba Case-Tool. The features of Real-time Java are described in Section 5 before the mapping from the Real-Time Statecharts to Real-time Java code is shown in Section 6. Section 7 draws a conclusion and gives a perspective on future work.

2. REAL-TIME MODELING

Developing software is divided in a phase of design and a phase of implementation. During the design phase, the structure and the behavior of the software are specified by appropriate modeling languages such as UML, constituting a standard for modeling these different aspects. A developer can use Object- and Class-diagrams to model the structure and Statechart- and Activity-diagrams to specify the behavior.

For the specification of software for embedded real-time systems, a number of object-oriented approaches [14, 1, 5, 7] including ROOM [14] have been proposed. As most ROOM concepts have been integrated into the UML 2.0 proposal of the main tool vendors [13], they are likely to soon become a part of the standard UML and will therefore be widely available. However, these concepts do not address the temporal behavior of the operational model and therefore do not improve the situation when it comes to the automatic code generation.

Another thread of development is the *UML Profile for Schedulability, Performance, and Time* [12]. The profile defines general resource and time models which are used to describe the real-time specific attributes of the modeling elements such as schedulability parameters or quality of service (QoS) characteristics. Besides an abstract *logic model*, a more concrete *engineering model* can be specified by using these extensions. The engineering model is later used for the required model analysis and code generation. However, it remains an open question in the UML profile how all required details of the engineering model are determined.

Therefore, the current practice when building embedded software with hard real-time constraints is to specify the software on a high abstraction level, then to partition it to make it run on a real-time operating system in concurrent threads (usually without adequate analysis), to implement it, to test if the time restrictions hold and then usually to re-partition it. Repeating this cycle a number of times is usually very costly but mostly unavoidable.

Typically, an embedded software application consists of several concurrent running threads[1], often these threads are periodic. When the application is employed in the real-time domain, the time when each thread completes is of crucial importance. The longest acceptable duration until the completion of a thread is designated by the *deadline*. In order to accomplish a schedulability analysis, the so called *worst case execution time (WCET)* of every thread has to be known. This time characterizes the upper bound of the possible duration of the thread, if it is executed on the processor of the underlying computer system without preemption [10].

Unfortunately, UML Statecharts do not allow the integration of these important attributes. The only way to bring time into UML Statecharts is the use of the so called *After-* and *When-* constructs. These constructs can be used to model temporal behavior, but are not sufficient to specify real-time behavior [4, 6]. The underlying zero-time execution assumption cannot be fulfilled in a distributed setting, as the required side-effects as well as the emitting of messages always require some minimal amount of time. Another weakness is that a reasonable real-time semantics for Statecharts including the side-effects is therefore not possible.

Another approach for modeling temporal behavior are *Timed Automata* [8, 11]. This kind of automata can be used to specify real-time behavior in a well-defined manner, dependent on clocks, but is very restricted in the output.

Real-Time Statecharts combine the advantages of Statecharts with those of the Timed Automata and add some constructs and restrictions which allow the user to specify real-time behavior and generate proper real-time code that ensures the specified timing properties. This extension overcomes the limitation of Statecharts w.r.t. real-time systems by supporting a well-defined real-time semantics based on Timed Automata [6].

3. REAL-TIME STATECHARTS

In the first part of this section, the syntax and semantics of Real-Time Statecharts are explained informally. Section 3.2 introduces the problem of time conflicts.

3.1 Syntax and Semantics

Figure 1 depicts a Real-Time Statechart. It consists of states and transitions, like usual Statecharts. The states are extended –compared to usual Statecharts– with the following annotations: Time invariants, clock resets associated with `entry()`- and `exit()`-Methods, WCETs for the `entry()`-, `do()`- and `exit()`-Methods, and a period for the `do()`-Method.

Transitions are associated with events, guards, side-effects, time-guards, clock resets, priorities, deadlines, worst case execution times, and channels and events for synchronization.

Let the set of clocks be denoted by C, a clock by $t_i \in C$. A time invariant is of the form $\bigwedge_{t_i \in C} t_i \leq T_i$, $T_i \in N \cup \{\infty\}$ and delivers the point in time when the specific state has to be left via a transition.[2] If the invariant does not contain a clock $t_j \in C$, this part is assumed to be $t_j \leq \infty$ and is omitted in the graphical representation.

[1] *Threads* are often designated as *Tasks* or *Processes*

[2] For all time annotations the unit of time has to be the same. For reasons of readability, it is disregarded in the formulas.

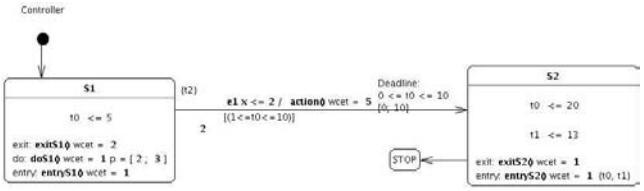


Figure 1: Real-Time Statechart

In Figure 1, the invariant of state S_1 is $t_0 \leq 5$ and the one of S_2 is $t_0 \leq 20 \wedge t_1 \leq 13$. The `entry()`-method is executed when a state is entered, the `exit()`-method before the state is exited. They have a WCET of 1 resp. 2 msec. Assigning clocks to these operations resets them at the moment of entrance resp. exit (t_0 and t_1 in state S_2). In order to perform analysises in the time domain (schedulability etc.), the WCET for each operation needs to be known (described by the annotation $\mathsf{wcet} = \ldots$). As the `do()`-operation is executed periodically while the automaton stays in the specific state, it is reasonable to annotate a period with this method. Often the user wants to specify a range for the period instead of a certain value. Thus the period for the `do()`-operation is specified by an interval ($p \in [2; 3]$ in State S_1).

A transition is triggered if the associated event (e_1) is available, the guard ($[x \leq 2]$) and the timeguard ($[1 \leq t_0 \leq 10]$) evaluate to true. The timeguard is of the form $\bigwedge_{t_i \in C} a_i \leq t_i \leq b_i$, $a_i \in N$, $b_i \in N \cup \{\infty\}$, $a_i \leq b_i$. Similar to the invariants, a timeguard is assumed to contain the expression $0 \leq t_j \leq \infty$ if no interval is specified for $t_j \in C$ and is omitted in the graphical representation.

When the transition fires, all clocks denoted in the set of clock resets are reset to 0 (t_2 in the example) and the sideeffect is executed (`action()`). When a transition is triggered, it fires. The firing will not be delayed. This behavior is generally denoted as *urgent* behavior.

If multiple transitions are activated, the one being triggered first fires. For the case that multiple transitions are triggered at the same time and that they are mutual exclusive, priorities have been introduced (In the example the priority is 2, a priority of 1 will be omitted in the graphical representation). *Only if multiple transitions, being mutually exclusive, are triggered at the same time, the one with the highest priority of these transition fires.* If the Statechart is in a parallel AND-state, it is possible that multiple transitions that are not mutually exclusive are triggered at the same time.

In addition to that, a worst case execution time ($\mathsf{wcet} = 5$) and a deadline are associated to a transition. The deadline is split into the relative and the absolute part. The relative part is of the form $[d_{low}, d_{up}]$ and describes that the switching (the execution of the transition) has to be finished at least d_{up} and at the earliest d_{low} after being triggered. The absolute part is depicted by a term of the form $\bigwedge_{t_i \in C} t_i \in [d^i_{low}, d^i_{up}]$ and describes the lower and upper bounds dependant on the clocks ($[0, \infty]$ is the default interval for both parts). In the example, the deadline is $[0, 10] \wedge t_1 \in [3, 11]$.

There exist 3 different types of synchronization: *External* synchronization by enqueing events, *internal* synchronization, and synchronization via *shared resources*. Internal

synchronization makes sense when the Statechart is in a parallel AND-state. Transitions can be associated with a channel and one of the actions `sending` or `receiving` (e.g. `a?` denotes `receiving` through channel `a` and `a!` denotes the complementary action `sending` through `a`). A transition associated with such an internal synchronization fires just when a transition with a complementary action through the same channel is triggered, too. The third possibility is the synchronization via shared resources. A shared resource is, for example, a specific memory area, that is written by some operations and read by some others. When some concurrent firing transitions are accessing the resource at the same time, the effect of priority inversion [10, 3] can appear and may delay the execution of one transition due to blocking effects.

Thus, the access should be controlled by a so called *monitor*, reducing the blocking time. The user is demanded to create a monitor-class in whose methods the critical sections are rolled out. When the worst case execution times of these methods are known and the monitor is associated with the operations that use the methods of the monitor to access the shared resources, then the maximal possible blocking time is considered in a scheduling analysis (see Section 4).

3.2 Timing analysis

The possibility to specify behavior, based on clocks, leads to the problem of time inconsistencies. If the user adds time annotations without care, it may lead to non-realizable behavior. Imagine a state with an invariant of $t_0 \leq x$ and leaving transitions, having a timeguard of the form $t_0 \geq x + n$, $n > 0$. So, this state cannot be left before its invariant exceeds, which will result in a so called *time stopping deadlock*. Analyzing the structure and the annotations of the Real-Time Statechart, these and other inconsistencies can be found by an algorithm.

There exist 2 different kinds of inconsistencies: A Real-Time Statechart is *inconsistent* if the specified behavior will definitely lead to problems. It is called *insecure* if it is possible, but not certain to run into problems. An example for insecureness is a state with a time invariant (e.g. $t_0 \leq x$) and leaving transitions which are triggered by events. As it is not possible to predict before runtime –without the use of a Modelchecker– if the according event occurs, it is possible for this Statechart to run into a time stopping deadlock, but not certain. A Statechart that is neither inconsistent, nor insecure is called *timeconform*. The different forms of inconsistency and insecureness are described in [4]. Some inconsistencies can be removed automatically by an algorithm, some can only be eliminated manually [4]. The user can choose on his own if he wants to use the algorithm for automatic inconsistency elimination.

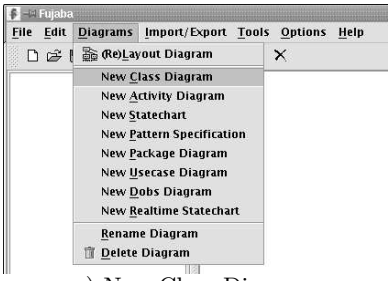
If the user intends to run multiple Real-Time Statecharts or different applications on the same target platform, he sometimes wants to specify a maximum processor load for every Statechart. This is done with the *utilization factor* (the utilization factor has a range from 0 to 1).

The sketched static analysis algorithm detects temporal inconsistencies at low costs. Some of the detected inconsistencies are removed automatically. Due to the incompleteness of the analysis, it is only a supplement to model checking but cannot, of course, replace it to detect all inconsistencies in the general case.

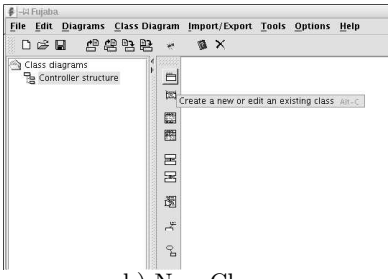
4. USER GUIDE

This section addresses users, new to Real-Time Statecharts. It gives a step-by-step instruction how to create a simple Real-Time Statechart, to check it for time consistency and to generate executable code.

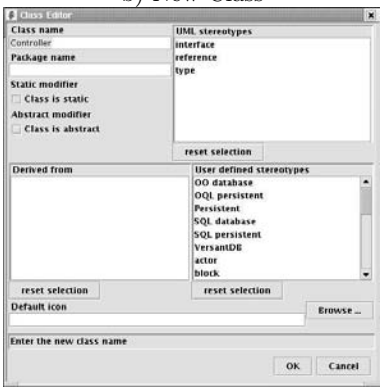
To use Real-Time Statecharts, the *Fujaba Tool Suite* and the Real-Time Statechart PlugIn are required.[3] After downloading, installing and starting Fujaba, Real-Time Statecharts can be used. As every Statechart is associated with a class, the first step before creating a Real-Time Statechart is creating a class diagram with at least one class (see Figure 2a) - c)). To do that, select the menu ,,Diagrams → New Class Diagram“ and enter a diagram name. After that choose the menu ,,Class Diagram → Create / Edit Class“ or press the first toolbar button as depicted in Figure 2 b). This will result in a dialog, where the name of the new class should be entered (*Controller* in the example). When choosing the menu ,,Diagrams → New Realtime Statechart“ (see Figure 3) the new class can be selected as the base class for the new Real-Time Statechart.



a) New Class Diagram



b) New Class



c) Edit the new class

Figure 2: Creating a Class

[3] www.fujaba.de

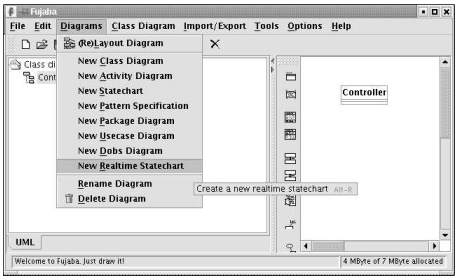


Figure 3: Creating a Real-Time Statechart

Figure 4 shows the diagram-specific menustructure and the toolbar, whoose actions can be used to edit the Real-Time Statechart.

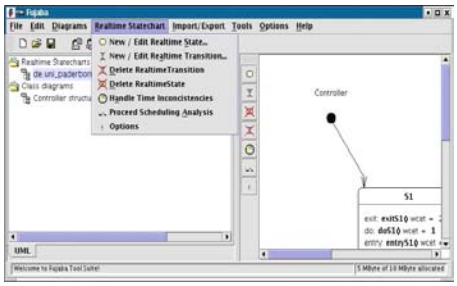


Figure 4: Menustructure and Toolbar

As a Statechart consists of states and transitions, the first two buttons on the toolbar represent the actions ,,New / Edit Realtime State...“ and ,,New / Edit Realtime Transition...“. There exist 3 different kinds of states: Start states, stop states and complex states. The start state should be unique for every Statechart. In the complex state, it is possible to specify the annotations described in Section 3.1 (see Figure 5).

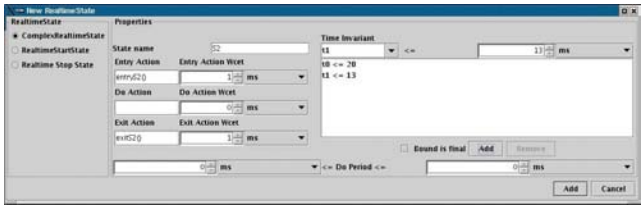


Figure 5: Dialog for States

The actions can be the call of methods, specified in the corresponding class diagram, or commands from the target language (e.g. Real-Time Java). Figure 6 depicts the dialog for transitions. For every timing attribute (deadline, invariant, timeguard), a final-flag can be set. When searching for time inconsistencies (see Sections 3.2) and an inconsistency that can be removed automatically by adjusting the specific attribute is found, this is only done if the flag is not set.

When creating new states or transitions, the user is asked to enter the worst case execution times for each operation. These declarations are used when searching for time inconsistencies (see 3.2) or performing a scheduling analysis. Additionally the WCETs of the ”simple, atomic operations”,

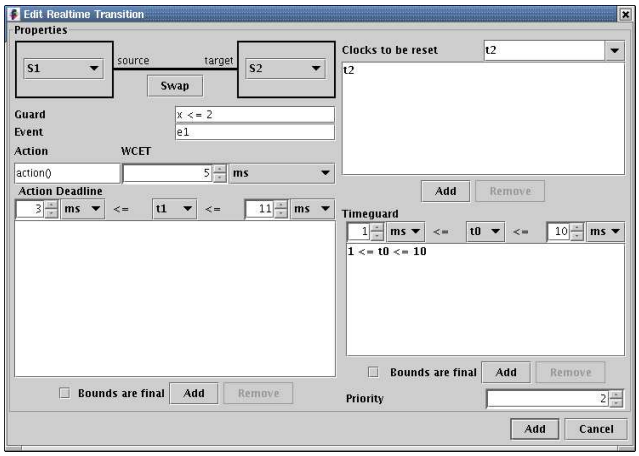


Figure 6: Dialog for Transitions

like the assignment of a long integer variable, the comparison of two integer variables and others, need to be specified in an xml-file (see Figure 7). Furthermore the WCETs of the actions can be specified in this doument, too. So in principle, it is possible to use external tools, determining the WCETs and providing this file.

```
<wcet>
  <action id="RealtimeStatechartactionTrans1"
     wcet="40" unit="ms" />

  <systemconstant name="INTEGER_ASSIGNMENT"
     wcet="1" unit="ms"/>
  <systemconstant name="LONG_INTEGER_ASSIGNMENT"
     wcet="1" unit="ms"/>
  <systemconstant name="INTEGER_ADDITION"
     wcet="2" unit="ms"/>
  <systemconstant name="INTEGER_COMPARISON"
     wcet="1" unit="ms"/>
  <systemconstant name="GET_METHOD_CALL"
     wcet="2" unit="ms"/>
  <systemconstant name="SET_OR_ADD_METHOD_CALL"
     wcet="3" unit="ms"/>
  <systemconstant name="OBJECT_ASSIGNMENT"
     wcet="3" unit="ms"/>
  <systemconstant name="TYPE_CAST"
     wcet="1" unit="ms"/>
  <systemconstant name="START_APERIODIC_THREAD"
     wcet="6" unit="ms"/>
  <systemconstant name="END_APERIODIC_THREAD"
     wcet="5" unit="ms"/>
  <systemconstant name="SLEEP_AFTER_THREAD_START"
     sleep="3" unit="ms"/>
  ...
</wcet>
```

Figure 7: wcet.xml

The path of the xml file can be set in the options dialog (see Figure 8). When generating code, a *schedule document* is generated (beside the source code files). This document contains information about all threads that are started, their WCETs and deadlines and which threads can run concurrently. These informations are needed for a scheduling analysis if multiple Real-Time Statecharts shall run in parallel on the same target platform. The destination path for creating the schedule document can be set in the options dialog as well.

In Section 3.2 it is described that a Real-Time Statechart can be timeconform, insecure or inconsistent. The radio buttons in the options panel are used to set the claimed security level. The last attribute that can be set is the utilization factor, described in section 3.2, either.

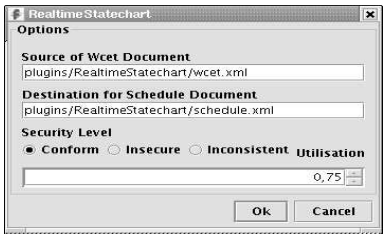


Figure 8: Options for Real-Time Statecharts

After specifying a Real-Time Statechart, the user should check for time conformity. This is done by calling the menu „Realtime Statechart → Handle Time Inconsistencies“ or pressing the equivalent button on the toolbar. Depending on the set security level and on the final-flags for the timing attributes (see Section 4), the inconsistencies are displayed and removed automatically.

When the desired degree of time consistency is achieved, it is possible to generate code. This is done by calling the menu „Import/Export → Export (All) Class(es) to Java“. Beside the source code file (that has the name of the class, associated with the Statechart), the schedule document is generated in the file specified by the options. This document is used for a scheduling analysis of multiple Real-Time Statecharts.

Even when there is just one Real-Time Statechart that should run on the target platform, the user should perform the scheduling analysis, as an additional class called *Main* is generated. The Main class contains code for assigning priorityes to the threads (important for scheduling) and starts all Real-Time Statecharts. As information about all involved Real-Time Statecharts is required, this class is not generated when calling „Export (All) Class(es) to Java“. For the same reason, the user is asked to add all relevant schedule documents after starting the Scheduling Analysis (see Figure 9).

After generating all target code, it can be compiled and started. For this, the `de.uni_paderborn.fujaba.umlrt.realtimestatechart.sdm`-package is required. This package is included in the `libs/sdm.jar`-file in Fujaba's PlugIn directory.

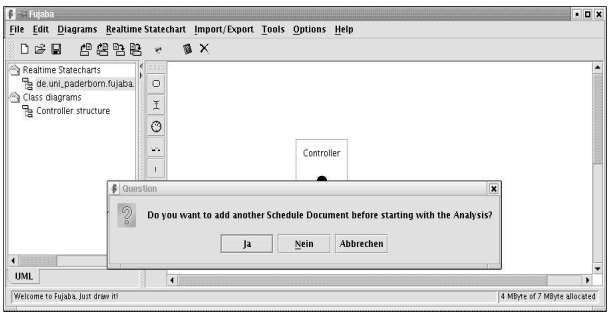


Figure 9: Declaration of Schedule Documents

5. REAL-TIME JAVA

The Real-Time Specification for Java (RTSJ) [2] provides an API, defining classes and methods, allowing the use of a real-time Scheduler and Memory Management in Java. In particular, it is possible to gain deterministic garbage collector behavior.

5.1 Scheduling

Figure 10 depicts the classes relevant for scheduling. As in every application exists at most one scheduler, the class `Scheduler` is implemented as a singleton. A Real-Time Operating System (RTOS) can provide plenty of different schedulers. One of the most common ones is the so called *Priority Scheduler*. To be conform to the specification, an implementation of the RTSJ has to deliver at least a Priority Scheduler. Of course it is possible to implement another scheduler (e.g. written in the programming language C) and use it in Java by accessing the routines of the scheduler via the Java Native Interface (JNI),[4] used in a subclass of `Scheduler`.

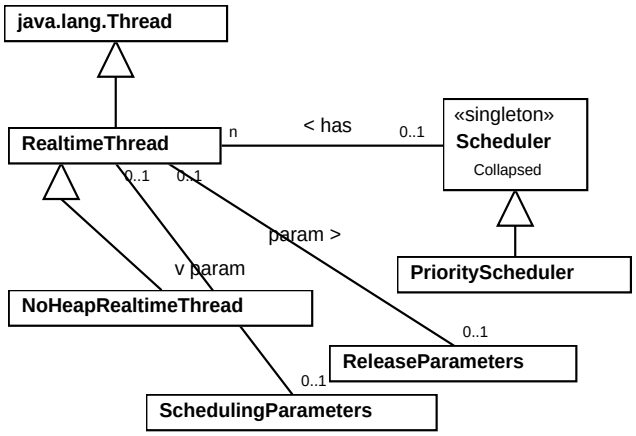


Figure 10: Scheduling

To assign the data relevant for scheduling like deadlines, periods, start time points etc. to a thread, the class `java.lang.Thread` is extended to `RealtimeThread`. Every Thread that shall be scheduled has to be registered with the instance of `Scheduler` via the `has`-association. Every `RealtimeThread` (or `NoHeapRealtimeThread`) can contain different parameters:

- `SchedulingParameters`

  This object contains just the priority of the associated thread, needed by the priority scheduler.

- `ReleaseParameters`

  The release parameters contain the WCET, deadline and the start time. If the thread is periodic, the object contains the period, too. If a deadline-miss-handler is specified, it is started at the point in time when the deadline is missed. Similarly, a cost-overrun-handler will be activated if the cost (WCET) is exhausted.

[4] java.sun.com

As it is not compulsory to provide the cost-overrun-feature, it is not available in all implementations of the RTSJ.

- `MemoryParameters and MemoryArea`

  When a `RealtimeThread` allocates new objects, they are allocated in the specified `MemoryArea` (see section 5.2). In the `MemoryParameter` object, a maximum amount of memory can be specified for the current and the immortal memory area (see section 5.2). Apart from this, it is possible to set an allocation rate in bytes per second that can be used for analysises.

- `ProcessingGroupParameters`

  On some real-time platforms, the operating system can guarantee that a thread never obtains more execution time than specified by the WCET. If the underlying operating system supports this capability and the real-time thread has a reference to a `ProcessingGroupParameter` object, the thread gets no more execution time than indicated by the cost attribute of this object.

- `java.lang.Runnable`

  If the real-time thread has a reference to a `runnable` object, the `run()`-method of this object is executed, after starting the thread, instead of the `run()`-method of the thread object.

```
public class PeriodicThread
 extends RealtimeThread
{
  public void run()
  {
    while (true)
    {
      ...
      waitForNextPeriod();
    }
  }
}
```

Figure 11: A periodic real-time thread

In Figure 11, it is shown how to implement a periodic thread. The method `waitForNextPeriod()` accesses the period attribut from the `ReleaseParameters` object and handles the coordination with the scheduler. The method is not exited before the scheduler allocates the processor for the specific thread, again.

5.2 Memory Management

Real-time Java divides the memory into different areas which are described in the following:

- Heap Memory

  The heap is the ,,traditional“ memory. Sun's virtual machines use nothing but the heap memory for dynamic memory allocation.

- Immortal Memory

  Instances allocated in the immortal memory area are never erased. It is usefull to place objects there existing during the whole time the virtual machine runs. Dynamic structures should not be allocated in this area.

- Scoped Memory

  Heap and immortal memory are implemented as singletons, but it is possible to create multiple scoped memory areas. A scope memory area has a so called *reference count*. This reference count is incremented every time a new instance is allocated and decremented when an instance is dereferenced. If a dereferenced object is placed in the heap memory, it will be freed and erased the next time the garbage collector runs. Being in a scoped memory, it will not be freed before the reference count for the whole scope drops to zero. At the moment when it becomes zero, the garbage collector is executed by the active real-time thread.

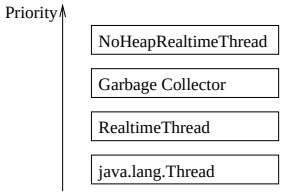


Figure 12: Priorities

In Figure 10 in Section 5.1, the `NoHeapRealtimeThread` is shown. Instances of this class have a higher priority than the garbage collector (see Figure 12). If the garbage collector runs and such a thread is started, it interrupts the garbage collector. Conversely, a `NoHeapRealtimeThread` will never be interrupted by the garbage collector. An ,,ordinary“ real-time thread has a lower priority than the garbage collector. So, real-time behavior can only be achieved with the use of these threads if the application abdicates on dynamic structures and the garbage collector is never started.

As mentioned above, every real-time thread is associated with a memory area. If a thread allocates new objects, they are allocated in its memory area.[5] Alternatively, objects can be placed in the different areas via the use of reflection. Therefore methods like `MemoryArea.newInstance(...)` or `MemoryArea.newArray(...)` exist.

Attention has to be paid when creating a `NoHeapRealtimeThread` object, as it is not possible to instantiate it on the Heap, and only within the context of a real-time thread.

This shall be clarified by the following example: In Figure 13, it is shown how to start a `NoHeapRealtimeThread`. At first a ,,help-thread“ `starter` is created. This object is allocated on the heap, but when it allocates objects on its own, they are placed in the scoped memory, represented by the object `scopedMemory`. This object `scopedMemory` *represents* a scoped memory area, but it itself is placed on the heap. The starter creates and starts a new `NoHeapRealtimeThread` (`nhrtt`). As the memory area of `starter` is `scopedMemory`, `nhrtt` is placed in `scopedMemory`. The RTSJ does not demand the `Thread.sleep(...)` instruction, but in the reference implementation[6] used, the virtual machine will hang up without this command. This is a bug, that probably is fixed in their new release.

[5] `java.lang.Thread` always allocates new instances in the heap memory

[6] We used the free available reference implementation from TimeSys (www.timesys.com)

```
public static void main(String[] args) {
  Starter starter = new Starter();
  ScopedMemory scopedMemory = new ...;
  starter.setMemoryArea(scopedMemory);
  starter.start();
}
public class Starter extends RealtimeThread {
  public void run() {
    NoHeapRealtimeThread nhrtt = new ...;
    nhrtt.start();
    try {
      Thread.sleep(3);
    } catch (InterruptedException e) {}
  }
}
```

Figure 13: Starting a `NoHeapRealtimeThread`

5.3 Monitors

Often concurrent running threads access the same memory. These *shared memory* areas have to be controlled by a so called *monitor*. Figure 14 gives an example for a monitor, implemented in Java. The key word `synchronized` avoids the parallel execution of the two methods.

```
public class Monitor {
  public synchronized int readDate()
  {
    ...
  }
  public synchronized writeData (int i)
  {
    ...
  }
}
```

Figure 14: Monitor

When a thread is inside a monitor (executes a `synchronized`-method of the object), it cannot be interrupted by another thread that wants to enter the monitor, too – even when this second thread has a higher priority. This leads to the well-known problem of priority inversion, described in [10, 3]. To deal with priority inversion, some useful protocolls have been developed. These protocols *do not* avoid the effect that a higher-priority thread is blocked by a lower-priority thread, but these protocols *minimize the blocking time*. Two well-known protocols are the *Priority Inheritance* and the *Priority Ceiling Protocols* [10, 3]. These protocols are represented in Real-time Java by two classes (see Figure 15). The priority inheritance protocol can be set when required.

The use of Priority Ceiling Emulation is also possible, but is only an optional element of any RTSJ implementation. It is important to mention that setting one of these protocols only has the expected effect if the underlying system provides the protocol.

6. CODE GENERATION

When generating source code from Real-Time Statecharts, at least one periodic thread –called *main thread*– is created. This main thread administrates the Statechart and has knowledge about the current state. In every period it checks periodically the outgoing transitions of the current state for being triggered and fires them if so.

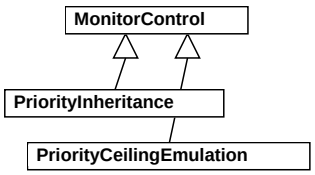


Figure 15: MonitorControl

There exist plenty of different patterns for implementing Statecharts. In order to satisfy real-time requirements, a variant without dynamic data structures has been choosen. In the target class, a constant is generated for every state (see Figure 16).

```
public static final int STATE_S1 = 0;
public static final int STATE_S2 = 1;
private int currentState = STATE_1;
public void handleTransitions() {
  switch (currentState) {
    case STATE_S1:
      if (/*check guard, timeguard and event*/) {
        action();
        t1 = 0;
        currentState = STATE_S2;
      }
      break;
    case STATE_S2:
    ...
  }
}
```

Figure 16: Implementation of a Real-Time Statechart

The attribute `currentState` indicates which state is valid so that the main thread just handles the (outgoing) transitions of the current state. As a Statechart can be in multiple states simultaneously because of hierarchy and parallelism, `currentState` is a more complex data structure than `int`, shown in Figure 16.

When executing the main thread, it runs on a real physical machine. Thus it cannot run infinitely fast, but with a minimal period. This results in the problem that a triggered transition is not recognized at the moment of activation, but with a delay, proportional –in the worst case– to the period. That's why the period should be as small as possible, but with respect to processor load and its own WCET, the period should be as long as possible. The deadline describes the point in time when a side-effect needs to be terminated, the WCET the time that is needed for execution and the period describes the delay between triggering and start of execution. Simplified, you can say: delay + WCET $\leq$ deadline must be satisfied. So the longest acceptable delay (and resulting, the period) can be determined, depending on the deadlines and WCETs. In general, shorter deadlines result in shorter periods.

Side-effects with a WCET greater than the period cannot be executed by the main thread. These actions are rolled out into aperiodic threads. The sequence diagram in Figure 17 depicts the application flow, when this is done. `FMainThread` is taken from the `sdm`-package, delivered with the Real-Time Statechart PlugIn. Its logic is the same for every Real-Time Statechart. The generated handler class implements the

interface `FRealtimStatechart` (defined in the `sdm`-package, too). It contains all the Real-Time Statechart specific information. First, the main thread calls `handleTransitions()` to determine the points of activation for all outgoing transitions of the current state. If the first activated transition has to be executed in an own aperiodic thread, the handler calls `executeAperiodicThread()` on the main thread to create and start a new aperiodic thread. The start is aperiodic, which results in concurrent running threads. As threads, that once terminated, cannot be reused in Java, the aperiodic thread is created in a scoped memory that frees the allocated memory after termination. After handling the transitions, the main thread waits for the next period.

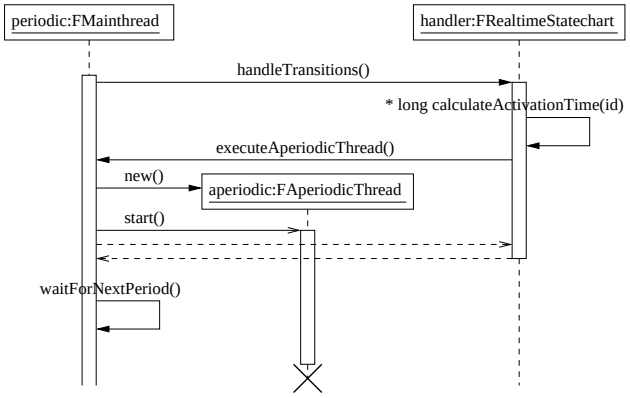


Figure 17: Logic of the main thread

The outlined scheme for the code generation permits to generate code for Real-time Java which ensures that the specified deadlines are always met when the WCETs are correct upper bounds.

7. CONCLUSION AND FUTURE WORK

Real-Time Statecharts provide a modeling technique, that allows to specify complex behavior on the one hand and real-time behavior on the other hand. Contrary to many other models, Real-Time Statecharts contain all information needed for code generation, which guarantees the specified timing characteristics [4]. The concepts for code generation are general, so that it is easy to adapt the generation algorithm to other target code. It is planned to provide a C++ generation in addition to the Real-time Java code generation soon.

Another capability of Real-Time Statecharts is the proof of temporal consistency with incomplete, but effective algorithms. In order to complete this verification, a model checker has been employed as reported in [9].

In addition, the visualization of recorded execution traces of Real-Time Statechart is planned [15].

Acknowledgements
We thank Florian Klein for his comments on earlier versions of the paper.

8. REFERENCES

[1] M. Awad, J. Kuusela, and J. Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion*. Prentice Hall, 1996.

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

[3] L. P. Briand and D. M. Roy. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. IEEE Press, 1999.

[4] S. Burmester. Generierung von Java Real-Time code für zeitbehaftete UML Modelle. Master's thesis, University of Paderborn, Software Engineering Group, 2002.

[5] B. P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. The Addison-Wesley Object Technology Series. Addison-Wesley, October 1999. Second Edition.

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

[7] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, January 2000.

[8] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *Proc. of IEEE Symposium on Logic in Computer Science*. IEEE Press, 1992.

[9] M. Hirsch and H. Giese. Towards the incremental model checking of complex real-time uml models. In *Proc. of Frist Fujaba Days*, Kassel, Germany, 2003.

[10] M. Joseph. *Real time systems : specification verification and analysis*. Prentice Hall international series in computer science. Prentice Hall, 1996.

[11] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Springer International Journal of Software Tools for Technology*, 1(1), 1997.

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

[13] OMG. UML 2.0 Superstructure final adopted specification. Technical Report ptc/03-08-02, August 2003.

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

[15] M. Tichy and M. Kudak. Visualization of the execution of real-time statecharts. In *Proc. of Frist Fujaba Days*, Kassel, Germany, 2003.

Towards the Incremental Model Checking of Complex Real-Time UML Models

Martin Hirsch and Holger Giese*
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[mahirsch|hg]@upb.de

ABSTRACT

Today, the verification of complex distributed embedded real-time systems employing model checking is largely limited by the state explosion problem. We first report on the current tool support for an approach which addresses this problem by means of a compositional model checking approach for a pattern and component based UML 2.0 designs. However, the current checking covers only an abstraction of the employed Realtime Statechart semantics (cf. [4, 9]), and the compositional approach only works for properties which refer either to a single pattern or a single component. We then present plans for an improved tool support which supports the full Realtime Statechart semantics, enables the compositional checking of non-local properties, and a model checker integration which triggers required checks after a model update automatically in the background.

Keywords

Model Checking, UML, Real-Time, Embedded Systems, Mechatronic Systems, Compositional Verification, Fujaba.

1. INTRODUCTION

Mechatronic components [6], which beside their local control of equipment are also interconnected with each other, result in a complex distributed embedded real-time system. As such mechatronic systems often contain real-time and safety-critical requirements, a proper approach for the real-time and safety analysis is mandatory (cf. [11]). The worst-case real-time characteristics w.r.t. deadlines must be predictable, and appropriate means for the validation and/or verification are required. Thus, the engineer can judge whether the resulting system still contains safety hazards. However, the verification of these systems by means of model checking is often not possible today due to the state explosion problem.

In this paper we first report on the current tool support for an approach which is addressing this problem. It is restricted to the specific case of software controlling mechatronic systems (cf. [10]). It uses a domain specific pattern and component based approach that employs a subset of the UML 2.0 component model. The complex software systems are composed of domain-specific patterns. These patterns differ to some extent patterns introduced in standard literature e.g.[7]. In [7] patterns are each identified by classes, associations etc. In our context the considered patterns are also described by their ports resp. port-roles. In this manner using/instantiating patterns we have to deal especially with the port behaviour. Each of these patterns can be verfied individually. The complete component behavior is derived by a composition of these patterns. For a syntactically correct composition the verified pattern and component properties are also guaranteed for the resulting overall system behavior. However, currently only an abstraction of the Realtime Statechart behavior can be checked and properties, which can be compositionally verified, must be locally defined either for a single pattern or a component.

A first required step is to transform the Realtime Statecharts into the format of the model checker in a way that all aspects of the semantics are covered. Then, the result of the model checking will result in less false negatives, because a less abstract model, which reflects the real behavior better, can be checked.

To address non-local properties, the approach can currently employ the compositional nature of the original approach and check a proper subset of the overall system. We propose to automatically determine such proper subsystems to incrementally check whether a property holds by using a series of subsystems with increasing model size. If from the verification follows that the property holds, the compositional nature ensures that the property also holds for the complete system.

Finally, we can improve the model checking support by integrating the automatic checking of properties if any update has happened in the related UML model. Due to the compositional nature each single checking can usually be done in the background in an incremental fashion.

This paper is organized as follows. In Section 2 the employed pattern-based approach for the modelling of real-time systems with UML is sketched, and the related compositional approach for model checking the resulting UML models is described afterwards in the remainder of this section. Then, we describe the currently in Fujaba realized tool support (Section 3). To support checking non-local properties, we review in Section 4 the shortcomings of the current tool support as well as ideas which permit to incrementally extend the checked model until the property holds. In Section 5, the resulting requirements for realizing these concepts are summarized. The paper finishes with a final conclusion.

2. REAL-TIME UML MODELLING

In this section, we first describe the main elements such as patterns, connectors and components, which are used when modelling real-time systems with UML. In the second subsection, we explain the design steps which are necessary to get a safe real-time system.

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

2.1 Basis Elements

In our approach a pattern comprises of a set of roles that interact only via a connector. Every connector connnects the related component ports in the final system. In the instantiated system the ports refine the roles and the components synchronize the ports. We further have the restriction that for each pattern we have to specify a protocol automaton and invariants for each role. An overall constraint in form of e.g. a TCTL formula is also possible. While usually in untimed models the connector behavior is omitted, channel delay is of crucial importance for real-time systems and thus has to be addressed explicitly in form of an additional connector automaton.

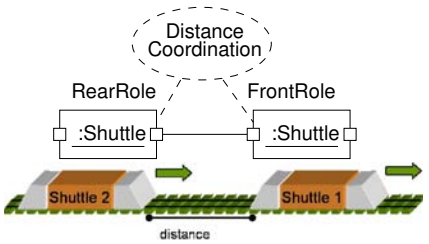


Figure 1: Modelling Example

Figure 1 shows an cut-out [10] of the convoy coordination between shuttles required for the software for the railcab research project[1]. The railcab system employs a passive track system and intelligent shuttles that operate autonomously and make independent and decentralized operational decisions. The vision of the railcab project is to combine the comforts of individual traffic, such as flexible scheduling, on-demand availability of transportation and individually equipped cars, with the cost and resource effectiveness of public transportation. The control infrastructure of the shuttle-based transportation system is based on satellite positioning and a wireless communication network that enables communication between shuttles and stationary installations.

In our example the "DistanceCoordination" pattern realizes the two roles "FrontRole" and "RearRole" which denote the positons in a convoy. The two roles are connected via a connector representing the communication network in the shuttle system. Components like the "Shuttle" component are designed by coordinating and refining each role automaton. The refinement has to respect the role automaton and additionally has to respect the guaranteed behavior of the roles in form of its invariants. An additional internal Realtime Statechart [9] for coordination is used to describe the required coordination. In the context of the example in Figure 1 this means that the shuttle must confirm to the "DistanceCoordination" pattern and has to operate as both "RearRole" and "FrontRole".

2.2 Design Steps

Because of the results and semantic definitions from [8, 10] we have the following sequence of integrated design and verification activities organized into the following 5 steps:

1. design the pattern and their roles,
2. verify each pattern,
3. design the composition,
4. verify each component, and
5. compose the system using the components and patterns.

Note that steps 1 and 2 have to be repeated for each required pattern. When step 3 and step 4 have already been performed with incomplete sets of patterns, the additional roles have to be added to the component automata. Step 5 finally ensures correct semantical composition by a correct syntactical composition. An additional 6th step to perform verification for the overall system after the composition made in step 5 is thus not necessary. The latter result is proven in [10]. This theorem is only valid for local properties for patterns or components.

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

3. CURRENT TOOL SUPPORT

So far there exists a version of the model checking plugin called UMLRTModelchecking for the CASE tool Fujaba [1] developed in the context of SHUTTLE PG project [2] at the University of Paderborn. This section describes its architecture.

3.1 Architecture

In SHUTTLE PG already some plugins helping us to model real-time systems with UML-RT were developed. Figure 2 shows an

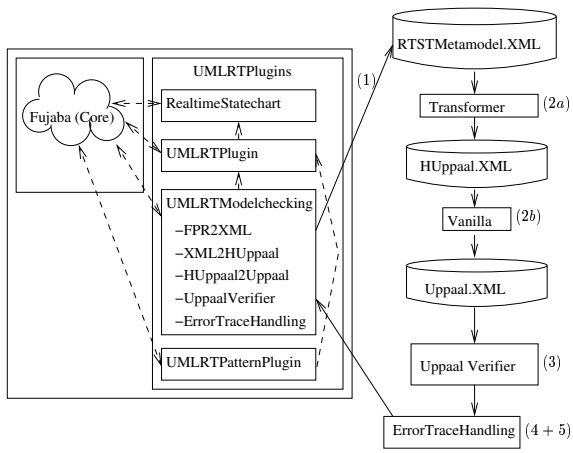


Figure 2: Plugin Architecture

overview of the plugins that are currently available. The UMLRTPlugin, which allows us to model Realtime Component Diagrams and generate JavaRT code, provides an own UML-RT metamodel. The Realtime Statechart plugin [4] is also integrated. It is used to model the role protocols and the component behavior. All other plugins depend on the UMLRTPlugin. The pattern plugin, named UMLRTPattern, enables us to identify patterns within the UML-RT model and to manage them in an own repository. It is further possible to select patterns form the repository and add them to the UML-RT model. Finally, there is a plugin named UMLRTModelchecking which enables the user e.g. to prove properties specified in the UML model as TCTL-formulas. In the following, this plugin is described in more detail.

3.2 Model Checking PlugIn

As seen in Figure 2 the UMLRTModelcheckig plugin requires the UMLRTPlugin and Realtime Statechart plugin. Further the plugin employs the real-time model checker UPPAAL [3] to verify whether the properties, specified in the UML model, are fulfilled. It is not necessary to add the whole model checker; only the verifier engine is required. Since the plugin execution is arranged in 5 steps, it is easy to use another real-time model checker like RAVEN [12] instead of UPPAAL by only making some changes.

The following paragraph describes the steps the existing plugin performs to check a UML model. (1) In the first step the meta data of a UML-RT model are exported and written to an separated XML-file. If any constraint in form of a TCTL formula has been

added to the UML-RT model, it is also written to the XML-file. If no TCTL-formula has been specified, the contraint "A[] not deadlock", which means checking the model for deadlock freedom, is automatically added to the file. (2a) Having generated this file, the plugin enables the user to transform the UML-RT model to Hierarchical Timed Automata [5]. Since there is no real-time model checker which works directly on the Hierachical Timed Automata model we have again to perform a transformation. (2b) The generated Hierarchical Timed Automata model is transformed to the flat Timed Automata model by using the the tool Vanilla [5]. The steps (2a + 2b) are executed in one step. Together with the TCTL-constraints defined in the original UML-RT model, the flat Timed Automata model can be used as input for the model checker tool UPPAAL [3]. (3) The UPPAAL verifier is automatically started after step (2). (4) The result of the verification and perhaps an error trace, if a property fails, is finally transformed back to UML view. (5) In order to make the result more descriptive, there is a visualization in Fujaba. Step (4) and (5) are currently only rudimentary solved by a text window showing the UPPAAL tool output. At present the transformation from step (2) works only for Realtime Statecharts. Our aim in this first version is to achieve a pessimistic abstraction of the original model. At present, only the hierachy and the basic structures like transitions, guards and states are supported, where as complex features like priorites are currently not supported. Because of the latter fact, we have much non-determinism in our Hierachical Timed Automata model and this enormously enlarges the state space. As mentioned before the basic structures are mapped to the basic structures of an Hierachical Timed Automata as defined in [5]. To realize the do-methods, exit-methods and entry-methods every state in a Realtime Statechart can have, we use a similar mapping as described in [9], where the semantic of Realtime Statecharts is introduced. Another problem is to transform the asynchronous communication model Realtime Statecharts have. In the Realtime Statechart model it is possible to send messages in an asynchronous way, if a transition turns (cf. [9]). We realize this by using the synchronous communication model of Hierachical Timed Automata and extend it by adding a separate automaton which manages a queue.

4. INCREMENTAL CHECKING

From [8] we know that local properties of patterns and components are preserved by composition and can be checked separately.

It is to be noted that if only the basic components and pattern that depend on the required property are checked, this can result in a false negative. If the verification of properties which involve more than one component is also required (non local properties), we have to slightly adjust the approach. In this case we can exploit the fact that proper subsets of the interconnected components and patterns can be seen as components and patterns again, if the local checks are not sufficient (cf. [8]).

Therefore, we can check a property by the following incremental procedure, shown in Figure 3. In the first step (1) we select a minimal composed pattern/component that contains all in the property referenced elements. After that we check whether the property holds (2). If so, we are done. Otherwise the set of considered elements has to be extended in a way, that the composed pattern/component contains the former one before continuing with step 2). Extending the model we embark on the follwing strategy. First we allocate all components specified in the selected requirements as mentioned before. If it is necessary to extend this model we first add thus components which interconnect the compontents selected in the last step, because often those compontents are involved in the considered context. Otherwise if there is no more

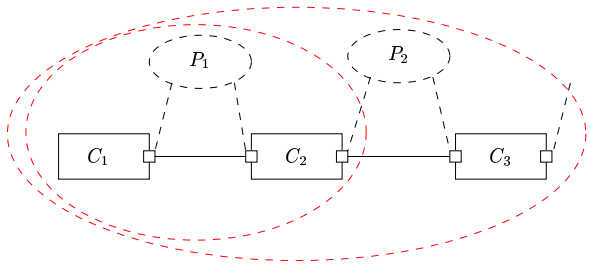


Figure 3: Example for incremental checking

component interconnecting another we select stepwise all neighbours and add them (cf. Figure 3).

5. PLANNED TOOL SUPPORT

In Section 3 we described the current tool support. The planned tool support is described in the following subsections.

5.1 Extended Transformation

Since there is no support for priorities which can be added to transitions in the Realtime Statechart at the moment, we extend the transformation by this concept. This is indeed not trivial, e.g. when transitions consist of synchronization elements, because it is not possible to invert them like guards. But if priorities are supported, we have a more precisely manner to specify our UML model, hence in the Hierachical Timed Automata model we have less non-determinism. Regarding to the state space using on-the-fly model checking, there is no enlargement as we have without priorities.

5.2 Improved Model Checking

In the current tool support there is no reasonable checking procedure for large models. Based on the idea of the incremental checking presented in Section 4 and the results from [10], we improve our checking procedure.

First, it should be possible to select a subsystem and check whether the specified proper TCTL-formulas are satisfied. It should also be possible to select TCTL-constraints in the original model and then start the verification process. To be sure, that the TCTL-formulas are syntactically correct, we will implement a syntax-checker. These two check procedures are finally combined with the incremental checking.

5.3 Background Checking

At present there is only one checking mode, namely "on request". For the extension we plan to add a "background mode". In detail this means that there will be something like a constistency mechanism. If there is any update in the UML model, the user will be informed and the verifier starts in the background.

6. CONCLUSION

The presented concepts, which the first author will realize within his master thesis, outline how to extend the compositional approach of [10] as well as its current tool support in several ways: (1) The foundation to model checking taking the full Realtime Statecharts semantics into account is sketched, (2) an incremental approach for checking non-local properties with the smallest sufficient subsystem model is presented, and (3) a plugin for the automatic checking of properties in the case of relevant model updates in the background is described.

We expect, that the compositional nature of the employed approach ensures that for such an improved tool support holds that the

size of the models which have to be checked due to model updates is usually rather small. Thus, the proposed background checking will usually be a feasible and convenient solution. However, currently no empirical evidence for this thesis can be presented and larger case studies are required to underpinn it by experimental data.

Acknowledgements

We thank Daniela Schilling for her comments on earlier versions of the paper.

7. REFERENCES

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

[2] http://www.upb.de/cs/ag-schaefer/Lehre/PG/SHUTTLE.

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

[4] S. Burmester. Generierung von Java Real-Time Code für zeitbehaftete UML Modelle. Master's thesis, Universität Paderborn, Deutschland, Fachbereich Informatik – Mathematik, Sept. 2002.

[5] A. David and M. O. Möller. From HUPPAAL to UPPAAL: A transformation from Hierachical Timed Automata to Flat Timed Automata. BRICS Report Series RS-01-11, University of Aarhus, Denmark, Deparment of Computer Science, BRICS, Mar. 2001.

[6] D. Dawson, D. Seward, D. Bradley, and S. Burge. *Mechatronics and the Design of Intelligent Machines and Systems*. Stanley Thornes, Nov. 2000.

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

[8] H. Giese. A Formal Calculus for the Compositional Pattern-Based Design of Correct Real-Time Systems. Technical Report tr-ri-03-240, University of Paderborn, Germany, Deparment of Computer Science, July 2003.

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

[10] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, Sept. 2003.

[11] D. S. Herrmann. *Software Safety and Reliability : Techniques, Approaches, and Standards of Key Industrial Sectors*. IEEE Computer Press, Nov. 1999.

[12] J. Ruf. Raven:Real-Time Analyzing and Verification Environment. *Journal on Universal Computer Science (J.UCS)*, (1):89–104, Feb. 2001.

Visualization of the execution of Real-Time Statecharts

Matthias Tichy and Margarete Kudak
Software Engineering Group
University Of Paderborn
Warburgerstr. 100
33095 Paderborn
[mtt|kudak]@uni-paderborn.de

ABSTRACT
Embedded software systems are used in nearly all of today's industrial products. Statecharts are used for the specification of the reactive behavior of those embedded systems. Since embedded systems have typically no rich user interface to display the current status of the system or even to display debug messages, another way to monitor the execution of the embedded system has to be used. In this paper we describe an extension of the Fujaba Tool Suite to support on-/off-line monitoring of the execution of Statecharts.[1]

Keywords
Statecharts, embedded systems, real-time systems, UML, Fujaba, monitoring, execution traces.

1. INTRODUCTION
Embedded software systems are a big factor in today's electronics or industrial products. Since a nearly failure-free operation of these embedded software systems is of utmost importance, high-level languages for the design and implementation of the embedded software system are employed. UML Statecharts are one of those high-level languages. They are used to specify the discrete behavior of software systems. Real-Time Statecharts [3] are a variant of UML Statecharts especially geared to the specification of hard real-time systems. Schedulability analysis and Java RT code synthesis are offered to omit error-prone manual implementation of the specification.

In case of a failure the developer of an embedded system wants to know what exactly has gone wrong in the system. Since embedded systems typically have no rich user interface to display its current state or to display debug messages, other means to view the behavioral activities, which lead to the failure, are required.

We propose a monitoring and visualization framework for UML and Real-Time Statecharts in the Fujaba Tool Suite [1]. This framework allows the developer to monitor the execution of the Statecharts. The monitoring data of the executed Statecharts are visualized using UML Sequence Diagrams and Real-Time Statecharts, with special markups. The visualization can either be used in on-line mode visualizing the current behavior of the system or in off-line mode visualizing older monitoring data. In this paper we focus on real-time embedded systems and Real-Time Statecharts. Nevertheless the approach is applicable to non real-time systems as well.

In the next section we give an overview of our approach. In Sections 3-5 we explain in more detail the different steps of our approach. We conclude in Section 6, describe the current state of work, and present some future research directions.

2. OVERVIEW

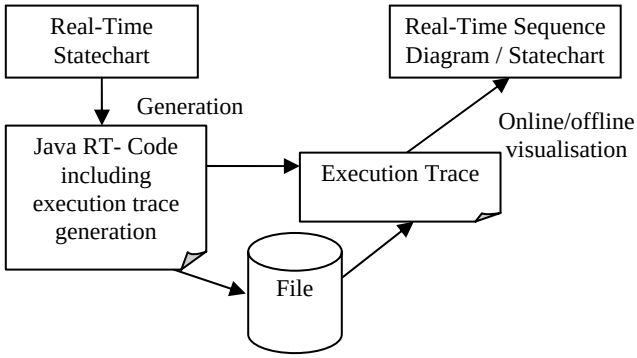


Figure 1. Framework architecture

Our proposed monitoring and visualization framework shown in Figure 1 consists of mainly 3 parts. We have to support the generation of execution traces to gather the data which contain the behavior of the executed Statechart. For each monitored Statechart an execution trace will be created during execution. In Section 3 we show in detail the contents of an execution trace and the different alternatives, which can be used to generate the execution trace.

Since typically more than one Statechart will be visualized, the execution traces of the different Statecharts must be merged prior to visualization. In Section 4 we give a brief overview of how we plan to merge those execution traces.

Finally, the merged execution traces are visualized by means of UML Sequence Diagrams and Real-Time Statecharts. UML Sequence Diagrams show the developer the message flow between the Statecharts as well as time annotations. Additionally, we add a graphical notion to show the current state of an executed Statechart. In the Statechart oriented view of the execution, the developer can see all Statecharts and their current state at a given time. Here, the developer may see the cause for a wrongly fired

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

transition. In Section 5 we give more details and some examples diagrams.

The above mentioned visualization of the Statecharts's execution can be done on-line respective off-line. That means, the visualization can display either past executions by reading the execution traces from a file (off-line) or display the behavior of currently executed Statecharts (on-line). If a monitored Statechart is changing its state very fast and very often, the visualization may lag in on-line mode.

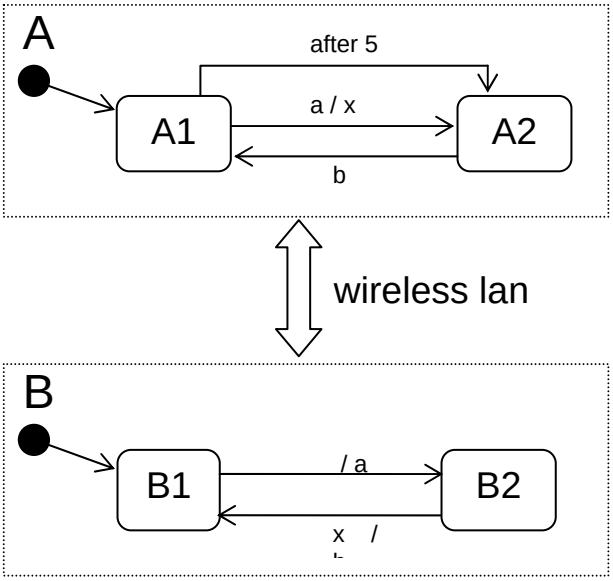


Figure 2. Example

In the example above (Figure 2) two Real-Time Statecharts are depicted which communicate via a wireless lan. In this small example it may be obvious (but not for larger ones), that both Statecharts may reach a deadlock. Initially Real-Time Statechart A is in state A1 and is waiting for the message a to change into state A2. Being in A1, A can also fire the transition "after(5)" and then go into state A2. Real-Time Statechart B changes from state B1 to B2 and sends the message a to A. Sending this message by wireless lan can take longer than 5 ms, so A may go into state A2 without having received the message from B. The consequence of this is, that Real-Time Statechart A is in A2 and is waiting for the message b, while Real-Time Statechart B is in B2 waiting for the message x to send the message b.

3. GENERATION OF EXECUTION TRACES

A trace of the execution of a Statechart consists of several execution activities. Each execution activity describes an activity of the Statechart, for example firing a transition.

Generation of such execution traces during runtime is time-critical, since it influences the temporal behavior of the Statechart. For hard real-time systems the goal is to minimize the temporal change.

For the generation of the execution traces several alternatives are available. At first the Java Debug Interface (JDI) [7] can be used to monitor the executed system. This would be a fairly easy solution, since no change of the monitored software system is required. Unfortunately, the runtime costs of JDI are way too high. The measurements in [6] indicate an increase of runtime by factors 100-300,000. Thus, JDI is completely out of the question.

A second solution is to instrument the compiled software. The Byte Code Engineering Library (BCEL) [8] can be used to change Java software after compilation. The negative effects of JDI can be avoided by using instrumentation. Though, it is difficult to find the correct code position where to add the data generation code. Since we are generating the source code for the Statecharts anyway, we may add the data generation code for the execution activities as well. Additionally, then the code for generation of the execution activities can be taken explicitly into account during scheduling analysis.

Additionally, we need to decouple the generation of the execution trace from everything else (file write, network write, etc.). We use a queuing approach. Each execution activity will only be written into a queue in the time-critical part of the system. During idle time, the queue entries will be read and further processed for writing to disk or network. If the queue becomes to large for the monitored system, the developer must take appropriate actions.

For an individual execution activity it is important to know what states, transitions and properties a Real-Time Statechart has. In the execution trace are mainly four activities listed: the change of a state, when a transition fires and the incoming and outgoing messages between Real-Time Statecharts, as can be seen in Figure 2. Additionally, an execution activity may include information about the different clocks used in the Real-Time Statechart and may include data about local variables which are used in transition guards.

4. MERGING OF DIFFERENT EXECUTION TRACES

A typical embedded system is comprised of more than one active system. These active systems are communicating with each other. For our approach we do not only need to generate and visualize the execution of one Statechart alone, but we have to display all concurrent Statecharts of the embedded system and their communication. Thus, the developer has the ability to see the interaction of the Statecharts. This interaction is a typical cause of complex errors in software.

To achieve this goal, we need to merge the execution traces of all involved Statecharts for a coherent view of the system. Embedded systems are often deployed on a distributed system and therefore are executed on different computers with different local clocks. Thus, clock differences and clock drift must be taken into account for the merging of the execution traces.

The differences between local clocks can be tolerated by the use of relative time stamps based on the known clock differences. Dealing with clock drifts is more difficult. If the clock drift is small and the absolute drift over the monitoring period is not too large (fraction of seconds), it may be possible to ignore it. Otherwise the approach of [4] has to be used. This approach uses causal relationships, e.g. between incoming and outgoing messages, to merge the execution activities in a correct order. In contrast to [4] we know the originating Statecharts and thus establishing a causal relationship is a lot easier.

5. VISUALIZATION

After merging the execution traces according to the last section, we can visualize them in two different ways.

5.1 Sequence Diagrams

UML Sequence Diagrams are used to display the communication between the participating Statecharts. Each Statechart is represented by an active object in the Sequence Diagram. The messages between the Statecharts are displayed as arrows between the lifelines. As an addition, the current state of the Statechart is shown as a special graphical object on the lifeline after a state change (see Figure 3). Time information has been added to the Sequence Diagrams to show the timing behavior. In real-time systems the timing behavior can often be the cause of problems, which are difficult to find. For the sake of a clearer presentation only some time annotations have been added to the figure.

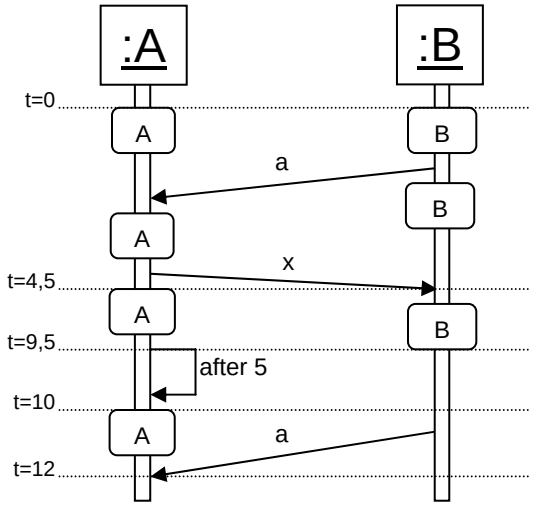


Figure 3. Sequence Diagram

The Sequence Diagram of Figure 3 shows the visualization of the Real-Time Statechart (Figure 2). Using this form of visualization the user can easily see, that Real-Time Statechart A fires the transition “after 5” and only afterwards receives message a from B. At this point of time the deadlock is reached, because A is waiting for message b while B is waiting for message x to send message b to A.

5.2 Statechart snapshots

For each point on the timeline of the above described Sequence Diagram a snapshot of all Statecharts can be displayed. Such a snapshot of a Statechart, displays the Statechart and an additional markup of its current state. The Statechart view may be easier for the developer to realize what the cause for a problem is. For example in this view the developer can see why a certain transition did not fire as expected and what the difference from expected behavior is. In Figure 4 the developer can see that the after 5 transition fired, since the message a has not been received.

5.3 Navigation

The user has different possibilities to navigate in the visualization. Using a timeline, he can slow up and speed up the visualization. If for example states are changing very quickly (fraction of seconds), it is not possible to see exactly what happened. Due to this, it makes sense to reduce the speed of the visualization. Another navigation possibility is to pause and resume the execution. Additionally, the user is able to specify a particular point of time and to start the visualization from this timestamp.

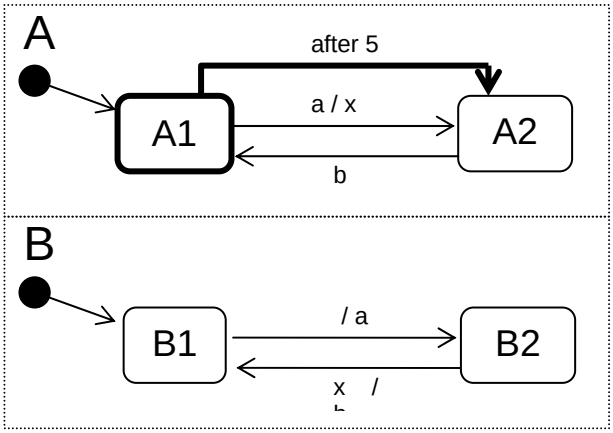


Figure 4. Real-Time Statechart with special markups

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a framework for the visualization of the behavior of embedded systems. This framework provides visualization in form of UML Sequence Diagrams and Real-Time Statecharts. The systems can be monitored off-line, i.e. a former execution is displayed. Additionally the systems can be monitored during runtime (on-line). Currently, we are at the implementation stage of the proposed framework.

We focus on real-time systems specified by Real-Time Statecharts. Nevertheless, the approach can be used to monitor non real-time Statecharts as well. The Fujaba Tool Suite includes support for StoryCharts [5]. StoryCharts are an extension of UML Statecharts which may include StoryPattern as do-activity. Our proposed approach may be extended to show the application of the StoryPattern as well (in [2] a related approach for the debugging of StoryDiagrams is described). Statistical evaluation of a number of execution traces may be used to discover potential faults of the system, which otherwise may go by unnoticed.

7. REFERENCES

[1] The Fujaba Tool Suite. http://www.fujaba.de, September 2003.

[2] Leif Geiger, Albert Zündorf. Graph Based Debugging with Fujaba. In Proc. of the Workshop on Graph Based Tools, International Conference on Graph Transformations, Barcelona, Spain, October 6 - 12 2002.

[3] Holger Giese and Sven Burmester. Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Computer Science Department, University of Paderborn, June 2003.

[4] C.E. Hrischuk and C.M. Woodside. Logical Clock Requirements for Reverse Engineering Scenarios from a Distributed System. IEEE Transactions on Software Engineering, 28(4):321-339, April 2002.

[5] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland, pp. 241-251, ACM Press, 2000.

[6] Katharina Mehner. Zur Performanz der Überwachung von Methodenaufrufen mit der Java Platform Debugger Architecture (JPDA). Java Spektrum, Ausgabe Nov./Dez. 2003 (in German).

[7] Sun Microsystems. Java Platform Debugger and Java Debug Interface. http://java.sun.com/products/jpda, September 2003.

[8] The Jakarta Project. Byte Code Engineering Library. http://jakarta.apache.org/bcel/, September 2003.

Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Metamodel

Pieter Van Gorp
Lab on Re-Engineering
University of Antwerp
pieter.vangorp@ua.ac.be

Niels Van Eetvelde
Formal Techniques in
Software Engineering
University of Antwerp
niels.vaneetvelde@ua.ac.be

Dirk Janssens
Formal Techniques in
Software Engineering
University of Antwerp
dirk.janssens@ua.ac.be

ABSTRACT

Increasingly more developers are applying refactorings - program transformations that can improve the design of existing source code - to make their software more easily adaptable to new requirements. Because small changes to object-oriented software (such as renaming a class) can require a lot of updates to several source files, tools that automatically update the affected files can save these developers a lot of time. Although refactorings are based on basic OO concepts (the redistribution of classes, variables and methods across the class hierarchy) only, today's development environments have hardcoded them on the abstract syntax trees of programming languages such as Java or C# and do not update middleware deployment descriptors. To facilitate the building of new refactorings and the extension of existing ones to new platforms, we suggest to implement refactorings as declarative specifications on a platform independent metamodel. This paper describes how the metamodel, the graph rewrite language and the architecture of the Fujaba UML tool can be extended to provide the required infrastructure.

Keywords

Refactoring, Metamodeling, Graph Rewriting, Model Transformation, Middleware, Code Preserver, UML, SDM

1. CONTEXT

A *refactoring* is defined as a "behavior preserving program transformation" [1]. Refactorings for OO software are based on the redistribution of classes, variables and methods across the class hierarchy, mainly for the purpose of facilitating future adaptations and extensions [2].

In order to maintain the system's integrity, a *refactoring tool* needs to update the source code references affected by a refactoring. It also needs to make sure that a refactoring is only executed when it is guaranteed not to introduce inconsistencies. Regarding the automatic updating of source code references, current generation refactoring tools do not take into account middleware deployment descriptors which obviously leads to deployment conflicts after refactoring. Furthermore, they give no formal guarantees on behavior preservation. Formal proofs rely on the correctness of the pre- and postconditions of the implemented refactorings. Existing refactoring implementations have hardcoded these constraints with third generation programming languages.

Fujaba is an open source UML CASE tool that was originally designed for Java code generation from Story Driven Modeling (SDM) specifications [3]. SDM is a visual programming language based on UML and graph rewriting. Graph rewriting is a feasible formalism to reason about the behavior preservation of refactorings [4]. In this paper, we report how we are extending Fujaba for implementing refactorings as SDM specifications on a platform independent metamodel without introducing inconsistencies in middleware deployment descriptors.

2. METAMODEL REQUIREMENTS

As Don Roberts explains in his Ph.D. thesis [5], building a refactoring tool involves more than implementing the program transformations. The tool should also be able to check the invariants, pre- and postconditions of a (sequence of) refactoring(s) to ensure source-consistency. Therefore, the tool needs a sufficiently expressive metamodel. Similarly, a developer may want to trigger refactorings based on the presence of the bad code smells they can solve. In this section, we evaluate whether Fujaba's metamodel is suitable for these purposes.

2.1 Evaluation of the Fujaba 4.0 Metamodel

Fujaba's metamodel consists of 2 layers of abstraction, physically separated by a lazy parser (see Figure 1).

The first layer is equivalent to the UML 1.4 metamodel which contains the coarse OO constructs (such as namespaces, classes, operations and attributes) but excludes all the method body information which is required for maintaining the consistency between the parsed model and the rest of the code when executing a refactoring transformation [6].

The second layer of Fujaba's metamodel refines the method body as a partial Java abstract syntax tree. Although the *if*, *for*, *while* and *assignment* constructs could be considered relatively platform independent (they occur in C++ and C# as well), they do not fit our refactoring purposes. On the one hand, one does not need to understand the difference between conditionals and loops: it only matters that a new variable scope is introduced. On the other hand, Fujaba's method body syntax tree does not contain the explicit *ac-*

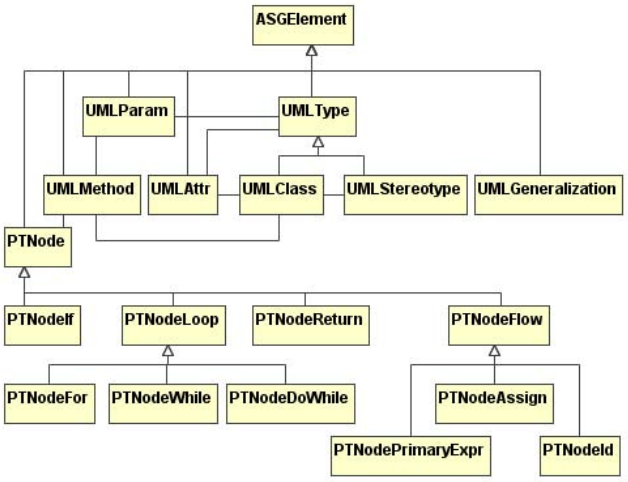


Figure 1: Fujaba 4.0 metamodel. *ASGElement* is equivalent to the standard UML *ModelElement* entity. We can reuse *UMLClass*, *UMLGeneralization*, *UMLAttr* and *UMLParam* from the first layer of this metamodel. All *PTNode* subtypes are part of the second layer. They reflect a partial Java abstract syntax tree and do not fit our refactoring purposes.

cess, *update* and *call* information that is needed to reason about refactoring.

Therefore, we propose to reuse the first layer of Fujaba's metamodel but use a new second metamodel layer that is minimal, yet adequate for refactoring.

2.2 Running Example

To illustrate the need for a metamodel extension, we highlight the tool requirements for automating a realistic Java middleware refactoring scenario. Our running example will be the "get cart items" operation from the shopping cart class from the EJB implementation of the open source xPetstore application [7]. Enterprise JavaBeans (EJB) is a standard component model for developing the application tier of a web application [8]. EJB components are managed by a container that interfaces with application server resources. Services such as object distribution, resource and transaction management and security are configured by specifying deployment attributes in an XML deployment descriptor.

```
1  /**
2  * @return Return a list of {@link CartItem} objects
3  *
4  * @ejb.interface-method
5  * @ejb.transaction-type
6  * type="NotSupported"
7  */
8  public Collection getCartItems() {
9    try {
10     ItemLocalHome ihome= ItemUtil.getLocalHome();
11     ArrayList items= new ArrayList();
12     Iterator it= _details.keySet().iterator();
13     while (it.hasNext()) {
14       String key= (String)it.next();
15       Integer value= (Integer) _details.get(key);
16       try {
17         ItemLocal ilocal= ihome.findByPrimaryKey(key);
18         ItemValue item= ilocal.getItemValue();
19         ProductValue prod= ilocal.getProduct()
20                                  .getProductValue();
21
22         CartItem ci = new CartItem(item.getItemId(),
23                                    prod.getProductId(),
24                                    prod.getName(),
25                                    item.getDescription(),
26                                    value.intValue(),
27                                    item.getListPrice());
28
29         items.add(ci);
30       }
31       catch (Exception cce) {
32         cce.printStackTrace();
33       }
34     }
35     // Sort the items
36     Collections.sort(items,
37                      new CartItem.ItemIdComparator());
38     return items;
39   }
40   catch (Exception e) {
41     return Collections.EMPTY_LIST;
42   }
43 }
```

2.3 Metamodel Extensions for Refactoring

2.3.1 Motivating Refactoring Scenario

Suppose we are reading CartEJB.java in order to better understand the system. Once we understand the undocumented try block from line 17 to line 29, we decide to extract its body into a new method with a name that fits the intention[1]. We will briefly share some details on the design of xPetstore and the EJB component model in general and then suggest a method name.

We know that the *_details* attribute represents a hash map from product key strings to the integer amount of items of that type in the cart. Line 17 uses the key of one product in the cart to retrieve a reference to the local item bean from the local item home object. Consequently, the home object checks whether the application server has a cached instance of the involved item and otherwise adds a new object instance for the item *ilocal* bean to an instance pool. The *ilocal* bean retrieves all its data from the underlying database unless it is configured with lazy loading. On line 18, a value object *item* encapsulating all this data is retrieved from the *ilocal* bean. Line 19 navigates to the product bean associated with the item bean and also constructs a value object for it. The data from these two value objects is then used from lines 22 to 26 to construct a value object representing the appropriate amount of cart items of the product from the current while loop iteration. Finally, on line 29, this value object is added to the list of items that our sample method is supposed to compose. What about extracting lines 17 to 27 to a method called *buildCartItemVOfromDB*? The while loop would then look like:

```
13     while (it.hasNext()) {
14       String key= (String)it.next();
15       Integer value= (Integer) _details.get(key);
16       try {
17         CartItem ci= buildCartItemVOfromDB(ihome,
18                                            key, value);
19         items.add(ci);
20       }
21       catch (Exception cce) {
22         cce.printStackTrace();
23       }
24     }
```

[1]This reengineering activity is commonly referred to as the *refactor to understand* pattern [9].

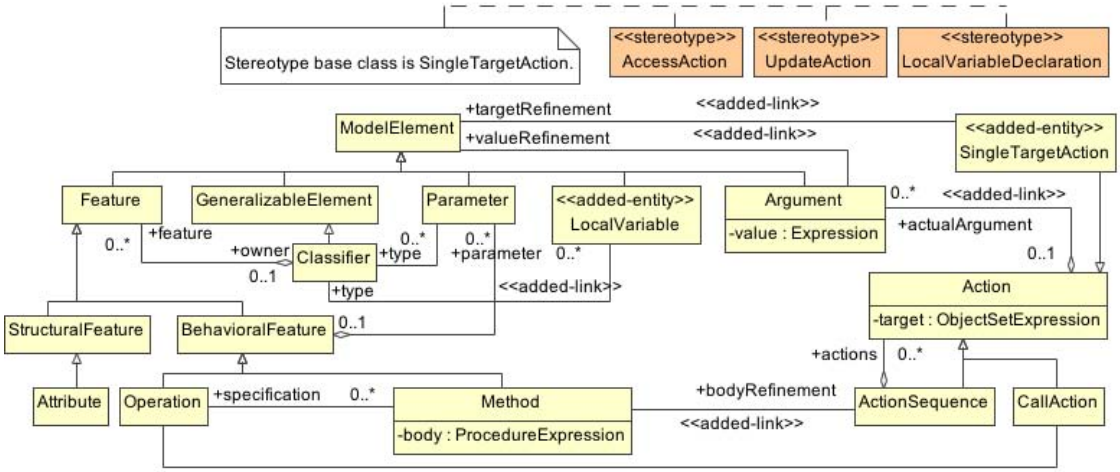


Figure 2: Overall view on the GrammyUML metamodel. The *added-entity* and *added-link* stereotypes highlight what is added to the UML 1.4 metamodel.

Note that we could move the declaration from *ihome* inside the try block as it is only used there, but we decide not to do it because retrieving the home interface of a bean is a resource-intensive operation.

2.3.2 Refactoring Implementation

Let us now consider the internals of this extract method refactoring. The precondition of this refactoring states that (i) the signature of the method that needs to be created may not result in a name conflict in the inheritance hierarchy of the containing class, (ii) there should be no return statement among the extracted statements, and (iii) within the selected source fragment, there should be at most one update to a local variable. The postcondition states that (i) there should exists a method with the chosen name in the containing class, (ii) each local variable that is used in the extracted fragment but not declared in it should be a parameter of the new method, (iii) each formal parameter that is used by the extracted statements should be a parameter of the new method, (iv) all checked exceptions that are thrown from the extracted statements (or from a method called by these statements), and that are not handled by the extracted statements, should be listed in the signature of the new method, and finally (v) in the original method, the extracted statements should be replaced by a method call to the new method.

To be able to express these constraints, our refactoring metamodel should include a *return action* (e.g. for precondition ii), an *update action* (e.g. for precondition iii), a *local variable declaration action* (e.g. for postcondition ii), an *access action* (e.g. for postcondition ii and iii), a *throw action* and a *handle action* (e.g. for postcondition iv), and finally a *call action* (e.g. for postcondition iv and v).

"Extract method" is an example of a refactoring that can only be executed by looking at the source code. It should be noted that this is not the cause of the need for all of our metamodel extensions. The "pull up method" refactoring for example can be triggered from a class diagram and also requires reasoning about method calls and accesses or updates to attributes. A part of its precondition for instance states that the method under consideration should not make any references (accesses or updates) to attributes that are part of the subclass. Line 12 of our running example accesses the attribute *_details* and calls the *keySet* operation for this object. Our sample method can not be pulled up because the superclass of *EJBCart* (i.e. *javax.ejb.SessionBean*) does not own or inherit the *_details* attribute. Without our metamodel extension, it would be impossible to check this precondition. Similarly, call behavior reasoning is required for another part of the precondition: no methods that are defined in the subclass should be called.

2.3.3 Proposed Extensions

GrammyUML is a minimal and backward compatible extension of the standard UML metamodel that is adequate to reason about refactoring (compose primitive refactorings, verify preservation of behavioral properties, and trigger refactorings based on code smells) [10]. As Fujaba's first metamodel layer is largely equivalent to the UML standard, all GrammyUML extensions apply to Fujaba as well. All primitives in the method body that are needed to reason about refactoring are modeled as variants of the standard UML "action" entity. *CallAction* is a standard UML construct that can be used to model call behavior in Fujaba. Accesses and updates can apply to attributes, parameters and local variables. The latter are not yet part of Fujaba's metamodel. We add a *LocalVariable* construct and associate it with its type. *LocalVariableDeclation* defines where a local variable is defined. Its scope reaches to the last *Action* of the containing *ActionSequence*. The *try* block from line 9 to line 39 for instance is an *ActionSequence* that implicitly defines the scope of *ihome*, *items* and *it*. Also note the anonymous local variables introduced by *_details.keySet()* on line 12 and *new CartItem.ItemIdComparator()* on line 37. Next to access, update and call behavior, we also need the notion of a return statement, and need to be able to reason about exceptions. Note that Fujaba's second metamodel layer would provide us a *PTNodeReturn* construct but it does not link this construct to the object that is returned. We provide this link in the *SingleTargetAction* construct whose *targetRefinement* can be reused by all its subclasses or stereotypes (*AccessAction*, *UpdateAction*, *LocalVariableDeclation*, *ReturnAction* and *ThrowAction*). Figures 2 and 3 summarize the proposed metamodel extensions.

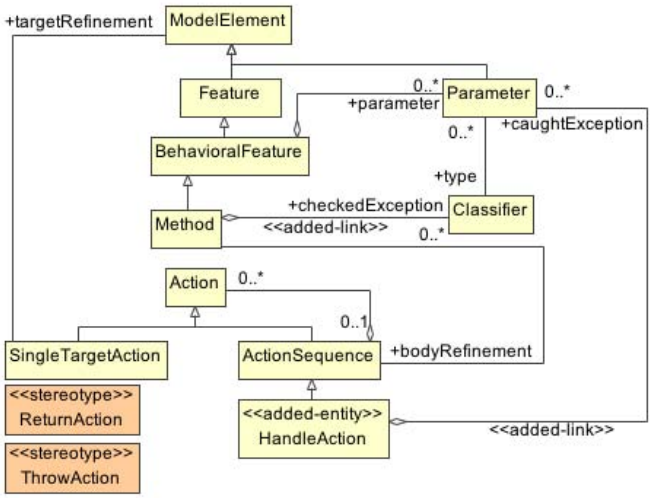


Figure 3: GrammyUML metamodel fragment to model exception throwing and catching and returning the flow of control from a method.

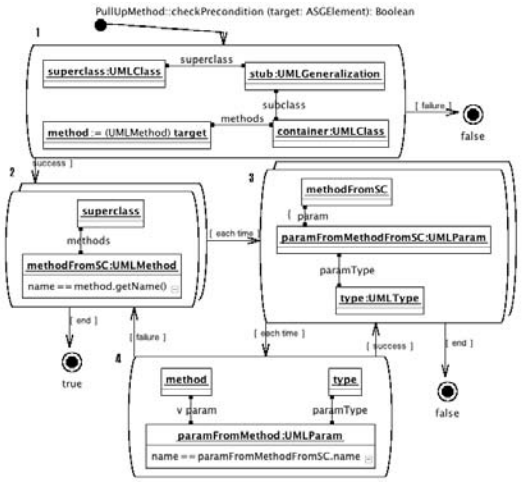


Figure 4: Story pattern for precondition (iii) of *Pull Up Method*

3. IMPLEMENTING REFACTORINGS IN SDM

In this section we describe a small experiment with Fujaba's graph rewriting language. For the sake of understandability, we concentrate on the relatively simple *Pull Up Method* refactoring.

3.1 Story Driven modelling

Story Driven Modelling (SDM) is a visual language for behavioral modeling based on UML activity diagrams, UML collaboration diagrams and graph theory [11]. Based on our small refactoring experiment, we identify some shortcomings of SDM for expressing our sample refactorings and suggest how this problem could be overcome.

3.2 Expressing Pull Up Method

From section 2 we recall that *Pull Up Method* has three important preconditions: (i) no references (accesses or updates) should be made to an attribute that is defined in the subclass, (ii) no methods that are defined in the subclass should be called and (iii) no method with the same signature should already exist in the superclass. Because of the lack of update and access information in the metamodel, we only implemented the third precondition. The story diagram expressing this precondition is shown in Figure 4. It has one parameter node: *method*, which is the method to be pulled up. The diagram consists of four patterns: In the first story, the unbound *superclass:UMLMethod* node, representing any class in the program is bound to the superclass of the class containing *method*. The three other stories represent a loop over each method *methodFromSC* in the superclass (story 2) and a comparison between the signature of *methodFromSC* and the signature of *method* (story 3 and 4). The signatures are compared by comparing the names and type of each parameter in the signature.

To be able to bind *paramFromMethod* to the correct parameter of *method* in story 4 (i.e. the parameter with the same index as *paramFromMethodFromSC*), we use SDM's *qualified associations*: the *param* link between *method* and *paramFromMethod* in story 4 is qualified with the index of the parameter. In our example, the value of this index is determined by evaluating the expression `paramFromMethodFromSC.getIndex()`. Although this constraint can be defined in the Fujaba environment, it is currently not possible to visualize it on a story diagram.

The control flow of the checkPrecondition procedure is straightforward: if the evaluation of the comparison is successfull, the next parameter is tried. The loop continues until all parameters are compared. If all parameters have identical names and types, the superclass contains a method with the same signature and the evaluation of the precondition returns false. If, however, the evaluation of story 4 fails, the method signatures are different, and the next method of the superclass is tried. This explains the *failure* transition between story 4 and story 2. We experienced that SDM currently does not support this 'nested loop break' construction, so for our experiment, we had to work-around this by implementing the comparison in pure java code.

If the precondition returns true, the execution step of the refactoring now moves the method from its containing class to the superclass. In a story diagram this is expressed by breaking the *methods* association link between the method and its containing class, and creating a new one with the superclass. SDM allows these graph rewriting operations by adding *create* and *destroy* modifiers to the associations between the nodes. The diagram for the refactoring is shown in Figure 5. To illustrate how Fujaba generates code out of the story diagrams, the java code of the execute story is given below.

```
1  public void execute(ASGElement target)
2  {
3    boolean fujaba__Success = false ;
4    Iterator fujaba__IterContainerRevSubclassStub=null ;
5    UMLClass container, superclass = null ;
6    UMLGeneralization stub = null ;
7    UMLMethod method = null ;
8    try
9    {
10      fujaba__Success = false ;
11      // check object is really bound
12      JavaSDM.ensure ( target != null ) ;
13      // ensure correct type
14      JavaSDM.ensure ( target instanceof UMLMethod ) ;
15
```

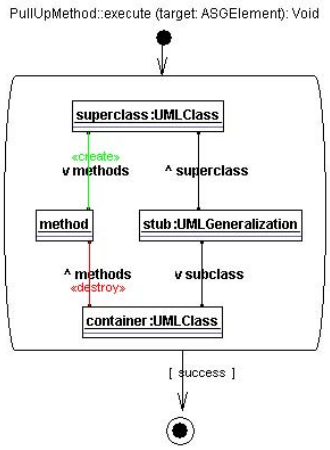


Figure 5: The execution story diagram of *Pull Up Method*

```
16      //  explicite type cast
17      method = (UMLMethod) target ;
18      // bind container : UMLClass
19      container = method.getParent () ;
20      JavaSDM.ensure ( container != null ) ;
21
22      // bind stub : UMLGeneralization
23      fujaba__IterContainerRevSubclassStub
24      = container.iteratorOfRevSubclass () ;
25      while ( !(fujaba__Success) &&
26      fujaba__IterContainerRevSubclassStub.hasNext ())
27      {
28       try
29       {
30          stub = (UMLGeneralization)
31          fujaba__IterContainerRevSubclassStub.next();
32
33          // bind superclass : UMLClass
34          superclass = stub.getSuperclass () ;
35          JavaSDM.ensure ( superclass != null ) ;
36          //  check isomorphic binding
37          JavaSDM.ensure
38          (!(container.equals (superclass))) ;
39          //  delete link
40          container.removeFromMethods (method);
41          //  create link
42          superclass.addToMethods (method);
43          fujaba__Success = true ;
44       }
45       catch (JavaSDMException fujaba__InternalException)
46       {
47       }
48      }
49   }
50   catch (JavaSDMException fujaba__InternalException)
51   {
52      fujaba__Success = false ;
53   }
54  }
```

To complete the experiment, the java code from the diagrams was reused in a small gui plugin for Fujaba, which allowed us to apply the refactoring on a class diagram (see Figure 6).

3.3 Extending SDM for refactoring

Even with such a small experiment we found that it was not possible to express every feature of a refactoring in SDM. Because SDM supports *statement activities*, every problem expressing a constraint graphically can always be solved by implementing this constraint in pure java code. However, to be able to express refactorings in an

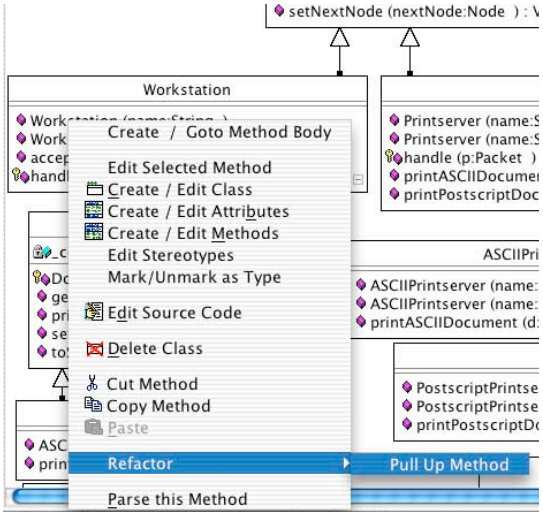


Figure 6: A refactoring plugin for Fujaba

efficient and elegant way, one needs to add some new features to the SDM language. In the next paragraph we will suggest an extension, which will be needed for expressing the sample refactorings in this paper, but might be useful for other applications too. Of course, more extensions will be needed in the future, as the implementation of other refactorings will raise the opportunity for adding new constructs to the language.

The extension we propose here is the possibility of using *Parameterized graph expressions* [4] in SDM. This is extremely useful for expressing pre- and postconditions of refactorings. For example: precondition (iii) of *Extract Method* requires a check that a local variable is updated only once inside the extracted block. This means that there will be at most one path from the ActionSequence node that represents the method body, to an UpdateAction node with the LocalVariable node as its TargetRefinement. To express this constraint using SDM, one would need an infinite number of stories. If we allow regular expressions on the links in a story diagram the expression

Figure 7: A regular expression pattern in SDM

would be sufficient to express all the possibilities. *MB* and *V* are nodes of type ActionSequence and LocalVariable, and *UA* is an UpdateAction node. When this graph expression is evaluated, *MB* and *V* are bound to to their respective parameters. Then a path between the two end nodes, that satisfies the regular expressions and unbound nodes (like *UA*) in the pattern, is determined. If no such path is found, the evaluation returns false. SDM already supports the definition of an arbitrary path between two nodes in a story. So this concept might serve as a basis for implementing the regular expression extension.

4. ARCHITECTURAL REQUIREMENTS

To ensure that the parsed source code will be regenerated appropriately, two new components are required in the Fujaba architecture.

We call these new components the “Code Preserver” and the “Refinement Repository”. These components complement the lexer, the parser, the metamodel, and the code generator.

4.1 Code Preserver

4.1.1 Definition

A *Code Preserver* is a development tool component that stores all the required source code files from which a model is extracted in such a way that the complete system can be regenerated from a transformation of the input model. A code preserver does not require a metamodel of all system properties and can preserve the original code layout.

4.1.2 Motivation

First of all, for the sake of simplicity, we want to minimize the amount of extensions to Fujaba’s metamodel as much as possible (without sacrificing source consistency). However, if we want to regenerate arbitrary method bodies with a conventional code generator, we would need a metamodel that contains all syntactically possible source code constructs (cascaded method calls, local variable declarations, type casts, type checks, ...). Otherwise, some (fragments of) source code statements would get lost.

In addition to this problem, code generators assume a fixed code layout for all instances of a particular metamodel element. This is undesirable in the context of refactoring, where developers don’t want to lose their layout each time they execute a refactoring.

4.1.3 Other Applications

To manage the rapid evolution of (and the number of alternatives between) today’s middleware component models, we want to minimize the work to add support for new XML deployment descriptors to our refactoring framework. When developing an application to evaluate the performance of a new component model (for which there are no code generators available yet), one may want to execute refactorings to evolve from a small running example to a more realistic prototype. With a code preserver, it would suffice to extend the Fujaba parser to integrate the new source code syntax in our refactoring tool. Without a code preserver, we would also need to write a new code generation template.

With a code preserver, it would suffice to extend the Fujaba parser to handle the new source code syntax. Without a code preserver, we would also need to write a new code generation template.

Our xPetstore sample is developed with the open source xDoclet code generator [12] and the Poseidon UML tool [13]. xDoclet generates skeleton classes and the deployment descriptors for the EJB component model from JavaDoc-annotated domain model sources. Poseidon visualizes the domain model classes as UML class diagrams. Poseidon’s model is stored in an XMI file. Instead of parsing and updating the XML deployment descriptors themselves, we need to update the annotated java sources that define the input model for xDoclet. We can obtain the new deployment descriptors by deleting them and regenerating them with xDoclet. Additionally, to maintain consistency with Poseidon, we need to update its XMI file. This illustrates how a code preserver can facilitate the integration of a set of special-purpose UML tools: Fujaba takes care of generating model transformation code from graph rewrite rules whereas xDoclet takes care of generating middleware framework code from stereotyped class diagrams that are visualized by Poseidon.

The fragment below illustrates the structure of *xpetstore.xmi*:

```
<UML:Class xmi.id='a1936' name='CartEJB'...
  isRoot='false' isLeaf='false' isAbstract='true'...>
  ...
  <UML:Classifier.feature>
    ...
    <UML:Operation xmi.id='a2006' name='getCartItems'
      isSpecification='false' ownerScope='instance'...
      isLeaf='false' isAbstract='false'>
      <UML:ModelElement.taggedValue>
        ...
        <UML:TaggedValue xmi.id='a2008'...
         dataValue='@return Return a list of
         {@link CartItem} objects&#10;&#10;
         @ejb.interface-method&#10;
         @ejb.transaction-type&#10;
         type=&quot;NotSupported&quot;'>
          <UML:TaggedValue.type>
            <UML:TagDefinition xmi.idref='a91'/>
          </UML:TaggedValue.type>
        </UML:TaggedValue>
      </UML:ModelElement.taggedValue>
      <UML:BehavioralFeature.parameter>
        <UML:Parameter xmi.id='a2009' ...
          kind='return'>
          ...
        </UML:Parameter>
      </UML:BehavioralFeature.parameter>
    </UML:Operation>
    ...
  </UML:Classifier.feature>
  ...
</UML:Class>
```

For the sake of readability, some fragments are suppressed as three dots. A code preserver could help us to keep track of the model references (e.g. *getCartItems* in the fragment above) and preserve all other information (like *@ejb.interface-method* and *@ejb.transaction-type*) without including the information into a dedicated metamodel. Of course, we would still need to write (or reuse) an XMI parser but this would also be the case if we would use a conventional code generator.

4.1.4 Overall Architecture

Figure 8 visualizes how the code preserver builds dynamic code templates with lexer input while the sources are parsed into a model. After refactoring, these templates are used to rebuild the files for the updated model. The design and implementation of the code preserver are beyond the scope of this paper.

4.1.5 Closing Remark

We should note that the abstraction level of a refactoring metamodel depends on the set of supported refactorings. Moreover, there is no ultimate refactoring metamodel as new refactorings are continuously being discovered. The code preserver can help to defer the inclusion of a metamodel entity until its semantics is an explicit part of the problem domain (i.e. the supported refactorings in our case). For example, currently we have not included type checks and type casts into our metamodel. Thanks to the code preserver, we do support the update of the referred class name when the rename class refactoring is executed. If we would ever need to implement a refactoring whose refactoring contract explicitly makes use of type casts we would include a dedicated *type cast action* in our metamodel.

4.2 Refinement Repository

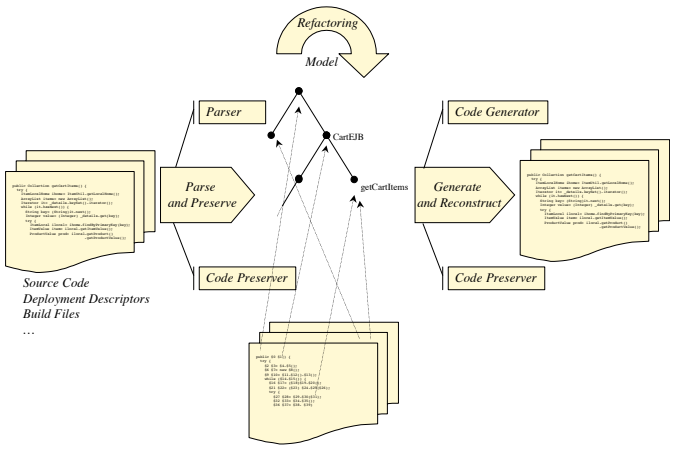


Figure 8: The role of the code preserver within the overall architecture.

4.2.1 Definition

A *refinement repository* is a development tool component that exposes the model to model refinement transformations of an MDA code generator.

4.2.2 Motivation

In this paper, we explore to what extent refactorings can be expressed on platform independent metamodels without sacrificing consistency with the underlying sources and configuration files. In the case of generated software, it is important to know the dependencies between domain entities and their derived component model specific classes.

First of all, refactorings on a domain entity should trigger a regeneration of all derived sources and configuration files. This regeneration can be implemented with existing black box code generators such as xDoclet.

Secondly, all manually written code that makes use of the generated classes needs to be updated as well [14]. Suppose, for example, we rename *Cart* to *ShoppingCart* in the problem domain of our xPetstore sample. Under the covers, this high level refactoring would be decomposed into the primitive "rename class" refactorings for *CartEJB*, *CartLocalHome*, *CartValue*, ... To execute such high level refactorings, a refactoring tool would need to query the code generator's "refinement repository" in order to learn about the name dependency from domain entities that are stereotyped as *EJB* to the name of model elements representing their bean class, local home class, value object class, ...

5. FUTURE WORK

First of all, this experiment calls for more validation. Among other things, we need to implement a GrammyUML parser for Java, a code preserver and a refinement repository. As a first step, we are extending the open source AndroMDA code generator with Fujaba's SDM. We have selected AndroMDA because of its standard JMI repository and its support for various middleware component models [15]. In this project, we will implement our suggested SDM extensions, along with lessons learned from a review of the MOF QVT submissions [16]. Another interesting project would be to extend Fujaba's lazy parser with the proposed metamodel extensions. We also have to investigate whether and how the UML 2.0 diagram exchange format can be parsed to GrammyUML because the current XMI standard does not include such information.

In our future work on Model Driven Refactoring with GrammyUML we may cover additional refactorings, additional formalisms and additional languages.

We are working on both additional primitive OO refactorings and high level composed refactorings supporting design and architecture evolution.

We are also evaluating how the emerging XQuery and XUpdate standards can be used to implement refactorings on XML representing GrammyUML models. Our goal is to compare our graphical (SDM), in-memory implementation (in Fujaba) with a textual (XML), database implementation mainly in terms of expressiveness and scalability.

An interesting validation case for the new code preserver architecture is implementing refactorings for C++ programs. Our approach would preserve not only the C++ programmer's code conventions concerning white-spaces and newlines, but would also preserve hand-written forward declarations across `cpp` and header files (which are often designed as API documentation).

6. ACKNOWLEDGEMENTS

We would like to thank Matthias Bohlen, the lead engineer behind AndroMDA, for his valuable feedback on the draft of this paper.

7. REFERENCES

[1] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[2] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. *Language Descriptions, Tools and Applications (LDTA)*, 2002.

[3] University of Paderborn Software Engineering Group. Fujaba. http://www.uni-paderborn.de/cs/fujaba/, August 2003.

[4] Tom Mens, Niels Van Eetvelde, Dirk Janssens, and Serge Demeyer. Formalising refactorings with graph transformations. *Fundamenta Informaticae*, 2003.

[5] Don Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[6] Sander Tichelaar Serge Demeyer and Patrick Steyaert. Famix 2.0 – the famoos information exchange model. http://www.iam.unibe.ch/ famoos/FAMIX/, 09 1999.

[7] Herve Tchepannou. xPetstore. http://xpetstore.sourceforge.net/java2html/xpetstore-ejb/xpetstore/services/cart/ejb/CartEJB.java.html, August 2003.

[8] Java Community Process. Enterprise JavaBeans specification, August 2003.

[9] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*, chapter 5, pages 103–107. Morgan Kaufmann, 2002.

[10] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In *Proceedings of the 6th International Conference on « UML » – The Unified Modeling Language.*, 2003.

[11] 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.

[12] xDoclet Project Team. xDoclet. http://xdoclet.sourceforge.net/, September 2003.

[13] Gentleware. Poseidon for UML. http://www.gentleware.com/, September 2003.

[14] Matthias Bohlen. AndroMDA 3.0 vision document: Moving to the agile world. http://www.andromda.org/developerdocs/, July 2003.

[15] AndroMDA Project Team. AndroMDA. http://andromda.sourceforge.net/, September 2003.

[16] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard, July 2003.

Integrated, Document Centered Modelling in Fujaba

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

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

Albert Zündorf
SE, Universität Kassel
Wilhelmshöher Allee 73
34121 Kassel
albert.zuendorf@uni-kassel.de

ABSTRACT

Originally, Fujaba is an UML based CASE Tool with emphasis on code generation and round-trip engineering. To provide better process support, we have developed the XProM plugin. The XProM plugin provides a document centered view on a project where all UML diagrams are embedded in dedicated chapters of an overall project handbook. The UML diagram and the corresponding document are integrated such that adding diagram(element)s automatically adds chapters for the description of these diagram(element)s.

1. INTRODUCTION

Many modern software development approaches propose a so-called usecase driven process, e.g. the Rational Unified Process RUP, [5]. In these approaches, requirements are analyzed using usecase diagrams and textual scenario descriptions. During the analysis phase these textual scenario descriptions are refined using UML behavior diagrams like sequence diagrams or collaboration diagrams. In the design phase, the program structure is defined using e.g. class diagrams and the program behavior may be modelled using e.g. statecharts. Unfortunately, processes like RUP define only the management aspects of software development. Technical guidance for the actual work and tight tool integration are still missing.

Adressing these shortcomings we devloped the Fujaba Development Process (FUP) providing technical guidance for the use of UML diagrams in different development phases and guidance for going from one process phase to the next. To provide optimal tool support for FUP, we extended the Fujaba case tool with an HTML based text document editor with integrated editing of UML diagrams, shown in Figure 1. Thereby, Fujaba provides a project handbook that guides the developers through the development process.

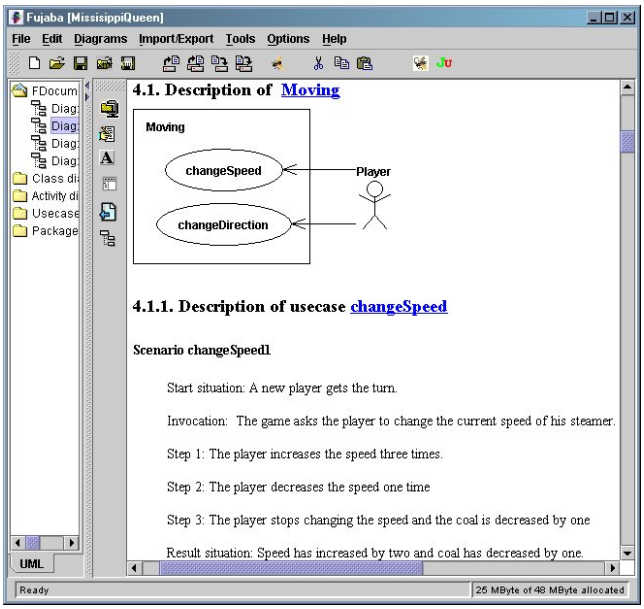


Figure 1: Usecase diagram with scenario for usecase `changeSpeed`

2. DOCUMENT STRUCTURE

When a new project is started, Fujaba loads a master copy file that provides an initial structure for the project handbook document. One may modify this initial document in order to adapt it to company or project standards and store the modified document as the new master copy for project handbooks.

Similarly, a template description file is loaded, that contains example document fragments for different kinds of UML (diagram) elements. When such an element is added to an UML diagram, the corresponding document fragment is copied into the project handbook. The new fragment is linked to the corresponding diagram element. For each diagram, a pre-defined (and modifiable) anchor describes where template fragments are inserted. The user may modify the template to adapt it to company or project specific needs in the same manner like the other project documents.

In the following we describe a possible document structure used in the software development laboratory course of Winter Term 2002 at the University of Kassel.

3. RUNNING EXAMPLE

The students had to model a game called "Mississippi Queen" using the Fujaba project handbook. In this game, every player has a steamer and has to ship it up the Mississippi collecting passengers. This example was given to the students as an initial example for a development process.

Every project starts with a skeleton of a project handbook. First the requirements are collected in usecases. By adding a usecase diagram to the document Fujaba automatically extends the handbook with a chapter for the description of this diagram. Similarly, adding a usecase to a usecase diagram automatically adds a template for the textual description of that usecase to the corresponding description chapter. These textual scenario descriptions have to be filled by the developer. Figure 1 shows a usecase diagram for the Mississippi Queen example and the textual description of a standard scenario for usecase `changeSpeed`.

Our usecase description template defines that each textual usecase description has a paragraph for the start situation, a paragraph for the invocation of that usecase, a number of steps outlining the execution of the usecase and a paragraph for the result situation.

4. AUTOMATISM

To provide further aid for transition from one process phase to another several automatisms have been introduced, cf. [2] or [3]. One of these derives a so-called story board from the textual scenario description. Initially, this story board is just an activity diagram with one activity for each element of the textual scenario. These activities contain the original textual descriptions as a comment. Now the developer is encouraged to model each step by a collaboration diagram that is embedded in the corresponding story board activity, cf. Figure 2.

The first activity of Figure 2 models the start situation of the `changeSpeed` scenario with a steamer s belonging to player Fred having color green. The developer modelled this situation as an object diagram consisting of a steamer object `s`, a game object `g` and an object `gui` representing the players graphical user interface. The attribute values of the object `s` are modelled as attribute pre-conditions, saying e.g. that the steamer still has six units of coal.

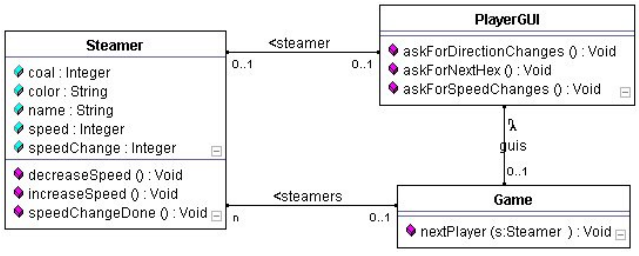


Figure 3: Class diagram automatically created during story boarding

During creation of story boards all used elements like objects, links, attributes and methods have to be provided with appropriate declarations in an accompanying class dia-

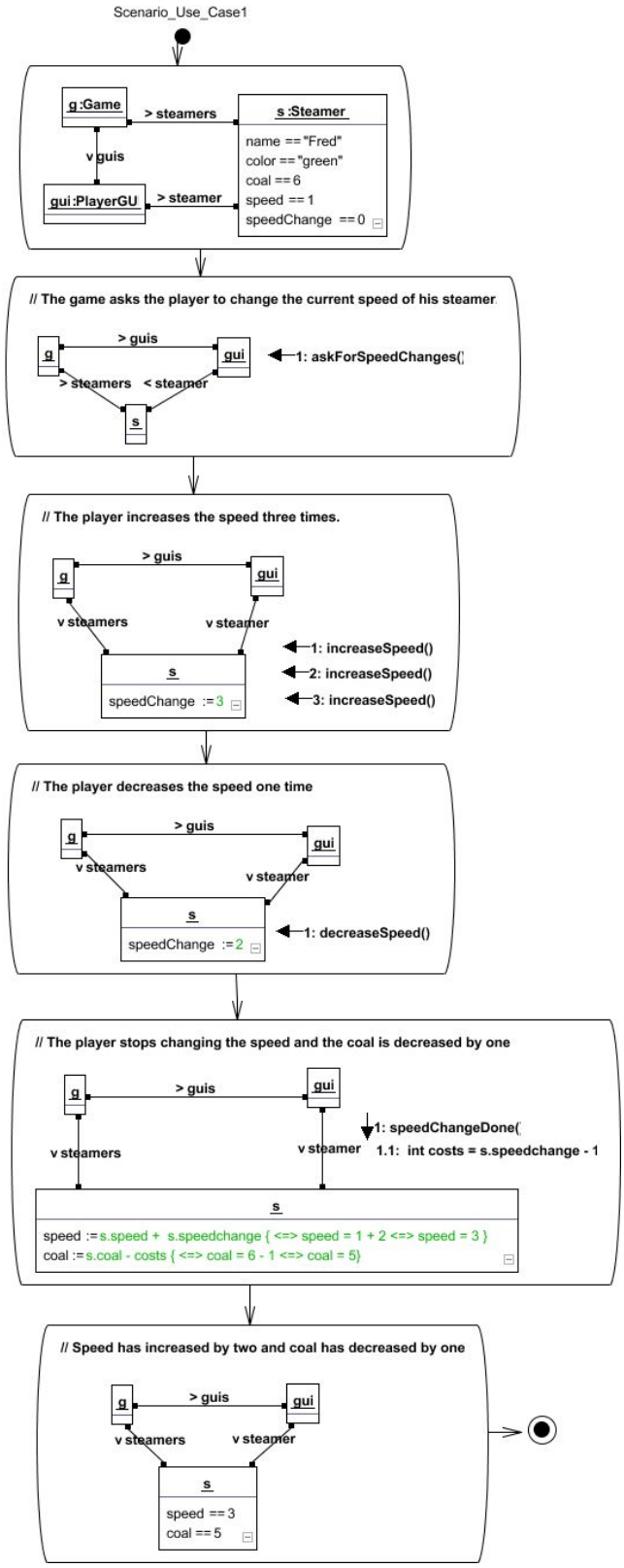


Figure 2: Storyboard for scenario `changeSpeed`

gram. This already ensures a consistent use of object kinds, attributes, links and methods throughout all scenarios and even within the following design phase. Figure 3 shows the class diagram of the Mississippi Queen example, that has been created during story boarding. Again, adding a class to a class diagram automatically adds a description chapter to the project handbook and adding a method to a class adds a template for the description of this method that usually contains the activity diagram modelling the behavior of this method.

Note, method bodies have to be specified, manually. However, in [2] and in [3] we propose a systematic approach for the derivation of method bodies from story boards.

In our work presented at [4], this is extended by automatic support for the generation of tests and for the embedding of test protocols into the project handbook.

5. CONCLUSIONS AND FUTURE WORK

In our approach, the concept of the initial project handbook structure and the automatically applied templates achieve a well structured and uniform project documentation.

The approach showed to work well for the students in the software development laboratory course. An idea of a software process was given by the initial structure of the project handbook and was supported continually by the evolving document.

This encourages us to specify a detailed software development process inspired by the Rational Unified Process, [5] and Extreme Programming [1] to be implemented as structured project handbooks. The developers can then be aided to apply the process by several automation tasks.

In addition, we plan to provide support for team collaboration and for project planning and project management.

6. REFERENCES

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Publishing Company, 1999.

[2] I. Diethelm, L. Geiger, T. Maier, and A. Zündorf. Turning collaboration diagram strips into storycharts. In *Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE 2002*. Orlando, Florida, USA, 2002.

[3] I. Diethelm, L. Geiger, and A. Zündorf. Uml im unterricht: Systematische objektorientierte problemlösung mit hilfe von szenarien am beispiel der türme von hanoi. In *Erster Workshop der GI-Fachgruppe Didaktik der Informatik*. Bommerholz, Germany, 2002.

[4] L. Geiger and A. Zündorf. Transforming graph based scenarios into graph transformation based junit tests. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*. Charlottesville, Virginia, USA, Septembre 2003.

[5] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Publishing Company, 1999.

Adapting FUJABA for Building a Meta Modelling Framework

Carsten Amelunxen, Alexander Königs, Tobias Rötschke, Andy Schürr
Technische Univeristät Darmstadt
Institut für Datentechnik, FG Echtzeitsysteme
Merckstraße 25
64283 Darmstadt, Germany
{amelunx|koenigs|rotschke|schuerr}@es.tu-darmstadt.de

ABSTRACT

The Real-Time Systems Lab performs research in the area forward engineering of automotive system software and reengineering of large industrial embedded systems in general. We need adequate CASE-tools to evaluate our approaches. These tools should be built on a shared meta modelling framework according to current standards (e.g. MOF 2.0, JMI, OCL, GXL). The FUJABA Tool Suite provides a substantial part of the required functionality (e.g. model editor, graph rewriting engine, Java code generator), but has to be adjusted to meet the standards. Currently, the necessary extensions can only be implemented by changing the FUJABA core, which is unsatisfactory. We would prefer to extract some FUJABA core functionality and distribute it over additional plug-ins. By doing so we could benefit from FUJABA for our meta modelling framework without compromising its present functionality.

1. INTRODUCTION

The mission of the Real-Time System Lab is to support meta model driven application development (MMDA) with a special emphasis on forward engineering of automotive system software and reengineering of large industrial embedded systems in general. The MMDA concept extends OMG's idea of model-driven application development (MDA) to the next higher level, where domain-specific tools and tool adaptions have to be developed as well. On this level meta tools are needed which support the specification of domain-specific modelling languages (including the adaption of general-purpose languages) and the generation of appropriate tools like editors, analyzers, and code generators.

As a consequence of this rather general perspective seven different research projects are currently under way, listed and categorized in table 1. Three projects belong directly to the automotive system software development area, whereas the remaining projects either address reengineering and safety issues of embedded RT systems in general or belong to different application areas. The entries of the table indicate on which levels of OMG's model hierarchy (M1-M3) system development activities are addressed:

M1 model driven application development

M2 development, adaption, extension, and integration of (domain-specific) modelling languages and tools

M3 development of meta modelling languages and tools for the model driven development of modelling tools

	Automotive			Civil Eng.	E-Learning	Med. Imaging	Security Eng.
Project	1	2	3	4	5	6	7
Construction	T	M		T		t	T
Transformation			T		a	M	
Analysis	T	M	T	T	a	T	
Aspect-Weaving							M
Integration		T	M	T	T	T	
Visualization				a	M	T	
Code Generation	M			a	a		M

M – Meta tool, tool, and application development (M1-M3)
T – Tool and application development (M1,M2)
t – Tool development (M2)
a – Application Development (M1)

Table 1: Topic Overview

All projects make contributions to all three modelling levels listed above, complementing each other. Due to the lack of space it is not possible to describe the specific goals of and relationships between all ongoing projects. As a consequence the following text describes only two projects in more detail which make very extensive usage of graph transformation techniques.

Evolution of complex embedded systems. Complex, software-intensive industrial systems like medical imaging systems and mobile switches evolve continuously and provide an increasing number of features. Usually, architectural models are proprietary with individual representations in design models and source code and are adjusted in isolated reengineering steps. Keeping architecture, detailed design and implementation consistent is difficult, as the architectural

consequences of small changes are not obvious to developers, and software architects can not manually check every detail. So continuous architectural analysis should be automated [18].

Data integration of CASE tools. Out industrial partner in the automotive sector uses several independent tools in different phases of the development process (e.g. DOORS for requirements engineering, Matlab for system architecture, CTE for testing), resulting in a variety of documents for the same project. Though these documents are related with each other, the tools cannot keep them consistent with each other. Trying to achieve consistency manually is time consuming and error-prone. So data integration of these tools should be automated [7].

Obviously, both projects need solution for meta modelling, which is also true for the other five projects. The general requirements of our projects are:

- generating repository interfaces, marshalling and unmarshalling tools from meta class diagrams
- generating static semantics checks and model analyzers from predicate logic expressions
- generating model transformation tools from graph rewriting rules
- generating tool integrators from triple graph grammars, a graph transformation based declarative approach to define document-boundary crossing consistency relationships [19]
- aspect weaving
- code generation

So it makes sense to define a shared meta modelling framework where all pieces fit together. In section 2 we describe how such a meta modelling framework should look like. Next, we explain in section 3 how FUJABA can help us to realize the framework and which modifications are necessary. In section 4 we explain how ECLIPSE should provide our framework with a common user interface and extensibility, and why FUJABA should adopt ECLIPSE as well. Finally, we discuss alternatives in section 5.

2. WHY WE NEED A META MODELLING FRAMEWORK

To address the specific issues mentioned in section 1 we need tools that provide different, but overlapping sets of features. As summarized in table 2, an intermediate investigation of existing tools reveiled, that many independent tools would be needed to provide the most important features. However, these tools do not interact with each other very well, and most commercial tools cannot be extended to match our requirements.

So we decided to select a small set of existing tools, i.e. FUJABA [2], ECLIPSE [17] and Dresden-OCL compiler [10] as a starting point for a meta modelling framework so that the tools can exchange data with each other. To increase the potential acceptance and interoperability of our framework, it should adhere to the most recent standards.

	FUJABA	SdP	Together	Artisan	ECLIPSE	Rational Rose	Rational Rose/RT	ArgoUML/Poseidon	Dresden OCL-Compiler
Architecture Description Language	-	o	o	o	-	o	(+)	o	-
Integration Framework	-	o	-	(o)	(+)	(o)	-	-	-
Extensible Code Generator	o	+	o	o	-	-	-	-	-
Model Driven Architecture	+	(+)	-	(+)	-	-	+	-	-
Meta Modelling	o	o	(o)	-	(o)	-	-	(o)	-
OCL-Compiler	-	-	-	-	-	-	-	+	+
Rule Interpreter	+	-	-	-	-	-	-	-	-
Available Source Code	+	-	-	-	+	-	-	+/-	+
License costs	+	-	-	(-)	+	(-)	(-)	+/-	+

Table 2: Tools and features for meta-modelling

The Object Management Group (OMG) is currently the authority in the field of standardization of (meta-)modelling languages. It is about to accept the current proposal for MOF 2.0 [14] as standard meta modelling language. The meta model of the next version of the popular Unified Modelling Language, UML 2.0, which is the OMG standard modelling language, will be defined using MOF 2.0. So it is very likely, that MOF 2.0 will be *the* meta modelling language and hence widely accepted in the near future.

The OMG standard format for tool interoperability is the XML Metadata Interchange 2.0 (XMI) format [16]. Many modelling and CASE-tools support XMI already. XMI used to come in two variants, UML-XMI and MOF-XMI according to the 1.x specifications of these languages. The variants will be replaced by MOF-XMI according to MOF 2.0, and hopefully be supported by more and more tools.

Rational Rose, the most important UML modelling tool, does already support MOF-XMI according to the old 1.x specification and will most likely support the 2.0 specification soon. Many other tools however, support only UML-XMI and our framework should be able to interact with these "legacy" tools. The University of the Federal Armed Forces Munich has already developed an XSLT script which can translate UML-XMI to MOF-XMI.

Many CASE tools including FUJABA are currently written in Java. To deal with meta models in Java, Sun as owner of Java has released the Java Metadata Interface (JMI) Standard [3]. As many tools will adopt XMI to exchange metadata, they will use the JMI standard to represent it internally. So our framework should be written in Java according to the JMI standard as well. The University of the Federal Armed Forces is working on a tool to convert MOF-XMI metadata to JMI compatible Java source code.

Static constraints that cannot be expressed graphically can be written using the Object Constraint Language (OCL),

the OMG standard for constraints [15]. However, the semantics of OCL is not fully defined though effort has been spent to provide it with more precise meaning, e.g. [8].

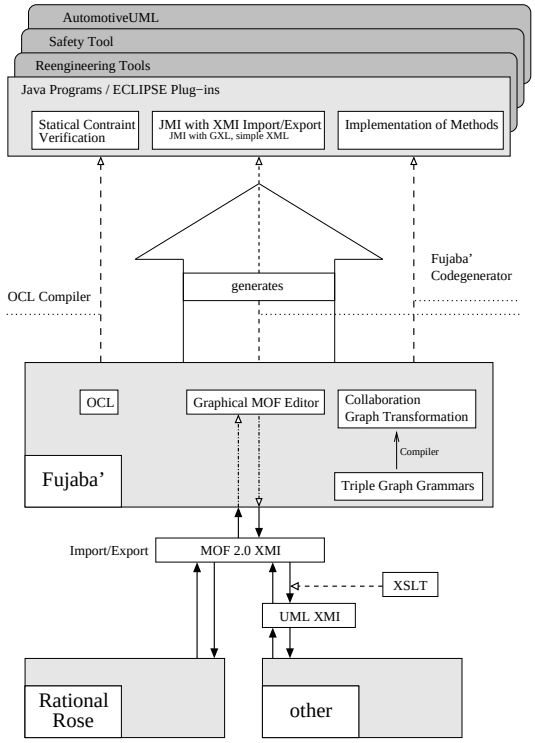


Figure 1: A vision of the meta modelling framework using FUJABA'

Figure 1 provides an overview of our framework denoted as FUJABA'. The lower part deals with the interoperability with other CASE tools using XMI. The upper part shows, how FUJABA' generates JMI compliant Java code for our meta modelling tools from MOF compliant models, graph transformation rules and OCL constraints.

3. FUJABA IN THE CORE OF THE META MODELLING FRAMEWORK

FUJABA Tools Suite 4.0 allows to create and edit UML class diagrams, activity diagrams and statecharts. Methods of classes defined in a class diagram can be implemented by specifying graph transformations on a UML collaboration diagram or just typing Java source code. The control flow is specified using UML-like activity diagrams, the graph transformations are embedded inside its activities. Because UML is widely known, it is easy to teach people how to use these so-called story diagrams.

Using this technique makes sense, as existing (meta) models are often defined in UML class diagrams. Mathematically, these can be described as graphs. As we want to reason about those existing models, graph rewriting rules provide a proper means to describe rules and transformations. Usually, graphically represented rules are easier to understand and mapped onto the existing models than textual rules. When graphically rules become too complicated to understand or are not expressive enough, the OCL still provides a textual means to fill the gap within the standard of UML.

Consistency rules between data models should be described by means of triple graph grammars. The FUJABA team is already working to integrate them into future releases.

There is still some work to be done to adjust FUJABA to our needs. The class diagramm editor has to be extended by a MOF 2.0 compatible editor. We need to propose, how MOF can be extended to cover story diagrams as standardized notation for graph rewriting rules. The code generator has to be replaced by a JMI-compliant variant. The XMI-JMI converter from the University of the Federal Armed Forces should be integrated into FUJABA. The existing path expression compiler should be replaced by integrating an OCL-Compiler, e.g. the Dresden OCL compiler. Optionally, FUJABA should be provided with UML packages, so that diagrams can be organized in a more efficient way.

Some of the extensions, like changes of the diagram editor and the code generator would require modifications of the FUJABA core. However, it would be better to realise this functionality as plug-ins, so that the user can choose between the original, initially more stable and the new functionality according to the new standards.

4. ADOPTING ECLIPSE

To provide our FUJABA-generated tools with a shared user interface and let them operate with other tools, we want to use ECLIPSE as integration framework. So all tools will be realized as ECLIPSE plug-ins. Additionaly, we plan to provide ECLIPSE with a MOF 2.0 compliant meta model.

If more and more FUJABA functionality is put into plug-ins, it makes sense that FUJABA adopts the ECLIPSE plug-in concepts as well. Eventually, FUJABA should become a MOF-compliant compiler and an editor, separately usable inside the ECLIPSE framework. Figure 2 shows, how the current FUJABA core should be transformed into a smaller FUJABA' core with more plug-ins that finally can be replaced by the ECLIPSE plug-in manager. This would allow more CASE tools to use its graph transformation engine, even if third-party editors are used. However, ECLIPSE does not provide data integration, but this is a core topic of our research.

5. RELATED WORK

Although not yet complete, we have considered some related work when working on our vision. Preliminary decisions for the choice of FUJABA and JMI over other alternatives are made based upon the following considerations.

5.1 IPSEN and PROGRES

IPSEN [13] is an existing integration framework, that could be a starting point for our framework. The graph rewriting system PROGRES [20], which is based on IPSEN, could be used as an alternative for FUJABA. PROGRES has a more powerful language than FUJABA and is more stable due to its higher degree of maturity. However, IPSEN and PROGRES are written in Modula-3 and do not run on Windows platforms, which are most common among our industrial partners. PROGRES uses a proprietary GUI and does

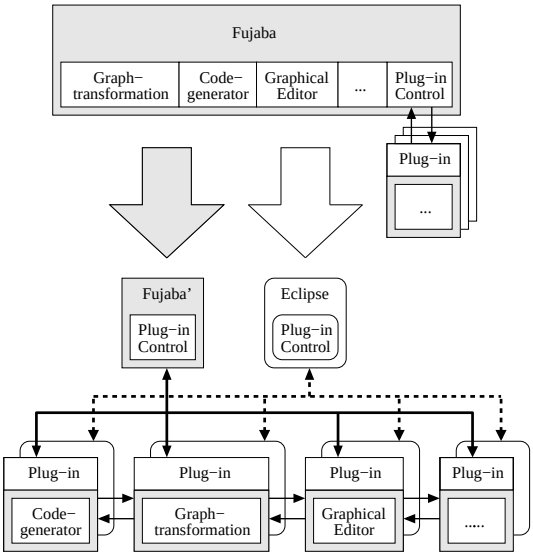


Figure 2: Evolving from FUJABA' to ECLIPSE

not comply to the newest standards, resulting in potentially less acceptance. Besides, IPSEN and PROGRES are very complex and do not have plug-in concept like FUJABA, so further extensions are more difficult to realize.

5.2 JAXB and XML Schema

The Java Metadata Interface (JMI) defines a mapping for MOF compliant models onto java technology. JMI uses XMI for the interchange of metamodel and metadata. A similar mapping could be achieved by using general data binding frameworks like JAXB [6], Zeus [4] or Castor [5] together with adequate XML schemata. The use of general data binding frameworks provides a less powerful mechanism because they are developed for common purpose applications as opposed to JMI which is dedicated to metadata. Due to the universal approach the entropy of such a mapping is higher than the entropy of a domain specific approach like JMI. JMI is a specialized interface and developed to fulfill the demands of a MOF mapping onto java technology. This is the advantage of JMI which makes JMI more suitable than a general data binding framework.

However, support for JAXB and XML Schema could be an optional feature of our framework. One possible application would be interoperability based on GXL [21] instead of XMI.

5.3 Meta CASE Tools

Available Meta CASE Tools like MetaEdit [12, 11], DOME [9], StP [1] would provide the possibility to generate CASE Tools using Meta Models. Unfortunately, they do not provide graph transformations and are usually closed source. So there is no way to extend those tools to meet our needs.

6. CONCLUSIONS

Adopting FUJABA for our meta modelling framework requires reasonable effort, but starting from scratch would be even worse. On the other hand, adjusting FUJABA towards the upcoming standards by using XMI-MOF 2.0 as exchange format, generating JMI-compliant Java code and integrating FUJABA with ECLIPSE will increase the interoperability and acceptance of FUJABA. As far as we can see, our proposals follow the current trend of FUJABA development. So joining the existing FUJABA community would be beneficial both for us *and* the community.

Our next steps will be a more detailed analysis of alternative approaches and a feasibility study, to find out the best strategy to realize our ideas.

7. REFERENCES

[1] Aonix. Software through Pictures (StP). http://www.aonix.com/stp.html.

[2] S. Burmester, H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. In *Proc. Workshop on Tool-Integration in System Development (TIS 2003)*, pages 51–56, Helsinki, Finland, Sept. 2003.

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

[4] Enhydra.org. ZEUS 3.5: Open Source Java/XML Data Binding. http://zeus.enhydra.org/.

[5] Exolab.org. Castor 0.9.5. http://castor.exolab.org/.

[6] J. Fialli and S. Vajjhala, editors. *The Java Architecture for XML Binding (JAXB), v1.0*. Sun Microsystems, Inc., Jan. 2003. http://java.sun.com/xml/jaxb/.

[7] R. Freude and A. Königs. Tool integration with consistency relations and their visualisation. In *Proc. Workshop on Tool-Integration in System Development (TIS 2003)*, pages 6–10, Helsinki, Finland, Sept. 2003.

[8] R. Hennicker, H. Hussmann, and M. Bidoit. On the Precise Meaning of OCL Constraints. In T. Clark and J. Warner, editors, *Advances in Object Modelling with the OCL*, volume 2263 of *LNCS*, pages 70–85. Springer, 2001.

[9] I. Honeywell. *DOME Guide, v5.2.2*, 1999. http://www.htc.honeywell.com/dome/.

[10] H. Hussmann, B. Demuth, and F. Finger. Modular Architecture for a Toolset Supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modelling Language. Advancing the Standard*, volume 1939 of *LNCS*, pages 278–293, York, Oct. 2000. Springer. http://dresden-ocl.sourceforge.net.

[11] S. Kelly. Improving the Integration of a Domain-Specific Modelling Tool. In *Proc. Workshop on Tool-Integration in System Development (TIS 2003)*, pages 57–60, Helsinki, Finland, Sept. 2003.

[12] S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE Environment. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *CAiSE*, volume 1080 of *LNCS*, pages 1–21. Springer, May 1996.

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

[14] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Proposal*, Apr. 2003. http://www.omg.org/docs/ad/03-04-07.pdf.

[15] Object Management Group, Inc. *Response to the UML 2.0 OCL RfP (ad/2000-09-03)*, Jan. 2003. http://www.omg.org/docs/ad/03-01-07.pdf.

[16] Object Management Group, Inc. *XML Metadata Interchange (XMI) Specification, v2.0*, May 2003. http://www.omg.org/docs/formal/03-05-02.pdf.

[17] Object Technology International, Inc. *Eclipse Platform Technical Overview, v2.1*, Feb. 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[18] T. Rötschke and R. Krikhaar. Architecture Analysis Tools to Support Evolution of Large Industrial Systems. In *Proc. IEEE International Conference on Software Maintenance (ICSM)*, pages 182–193, Oct. 2002.

[19] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1995.

[20] A. Schürr, A. J. Winter, and A. Zündorf. Developing Tools with the PROGRES Environment. In Nagl [13], pages 356–369.

[21] A. Winter, B. Kullbach, and V. Riediger. An Overview of the GXL Graph Exchange Language. In S. Diehl, editor, *Software Visualization*, volume 2269 of *LNCS*, pages 324–336. Springer, Berlin Heidelberg, 2002.

Layout Algorithms for FUJABA Diagrams*

[Extended Abstract]

Kalle Aaltonen
Department of Computer Sciences
University of Tampere
Kanslerinrinne1, FIN-33014,
University of Tampere, Finland
kalle.aaltonen@uta.fi

Jyrki Nummenmaa
Department of Computer Sciences
University of Tampere
Kanslerinrinne1, FIN-33014,
University of Tampere, Finland
jyrki@cs.uta.fi

Timo Poranen †
Department of Computer Sciences
University of Tampere
Kanslerinrinne1, FIN-33014,
University of Tampere, Finland
tp@cs.uta.fi

ABSTRACT

We describe how a layout algorithm for the UML class diagrams is designed and implemented. We also study how the algorithm can be used with the FUJABA and applied with other diagram types of FUJABA.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.2 [**Software Engineering**]: Object-oriented design methods—*class diagrams, layout algorithm*

General Terms

Layout algorithm, polyline grid drawing, FUJABA plugin

1. INTRODUCTION

The Unified Modeling Language (UML) [12] is currently the standard notation for modeling software-intensive systems. The UML can be used to visualize, construct, and document the artifacts of a software system.

Graph drawing [2] addresses the problem of constructing geometric representation of graphs. Every UML diagram can be though as a graph, and how a graph corresponding to a given UML diagram should be drawn, is a graph drawing problem.

Usability tests [9, 10] and many theoretical results (see [2], and references given there) verify that

> a good diagram helps the reader to understand the system, but poor diagram can be confusing.

Three fundamental issues in graph drawing are the *conventions* when drawing graph, the *aesthetic criteria* for a readable drawing and the *constraints* that a drawing may be required to satisfy.

A drawing convention is a special rule that the drawing have to satisfy [2]. Widely used conventions are the following: polyline drawing (each edge is drawn as a polygonal chain), straight line drawing (edges are straight lines), orthogonal drawings (each edge is drawn as a polygonal chain of alternating horizontal and vertical segments), grid drawing (vertices, crossings, and edge bends have integer coordinates), planar drawing (no edge crossings) and upward (down, left and right) drawing.

Constraints are used to provide semantic information about the meaning of the drawing in order to better reflect the features of the underlying model [5, 13] (for example: place "most important class" in the center of the layout). These types of instructions usually cannot be automatically deduced by a diagram layout algorithm. Hence, the user has to give them as additional input.

An *aesthetic criterion* is a general graphical property of the layout that we would like to have [1, 2, 8, 10]. A well chosen aesthetic criterion improves the readability of the given layout (for example: minimize the total number of edge crossings, maximize the symmetry in the drawing, and minimize the number of the edge bends in the drawing).

FUJABA is an Open Source UML CASE tool [3]. FUJABA project started at the software engineering group of Paderborn University in 1997. In this extended abstract we describe briefly the basic theoretical properties of the layout algorithm for class diagrams, how the algorithm is implemented for the FUJABA, and how it can be used as a plugin. Finally we discus open problems related on the layout algorithm. We also study the applicability of the layout algorithm for other diagram types used in FUJABA.

*Supported by the Academy of Finland (Project 51528).

†Work funded by the Tampere Graduate School in Information Science and Engineering (TISE).

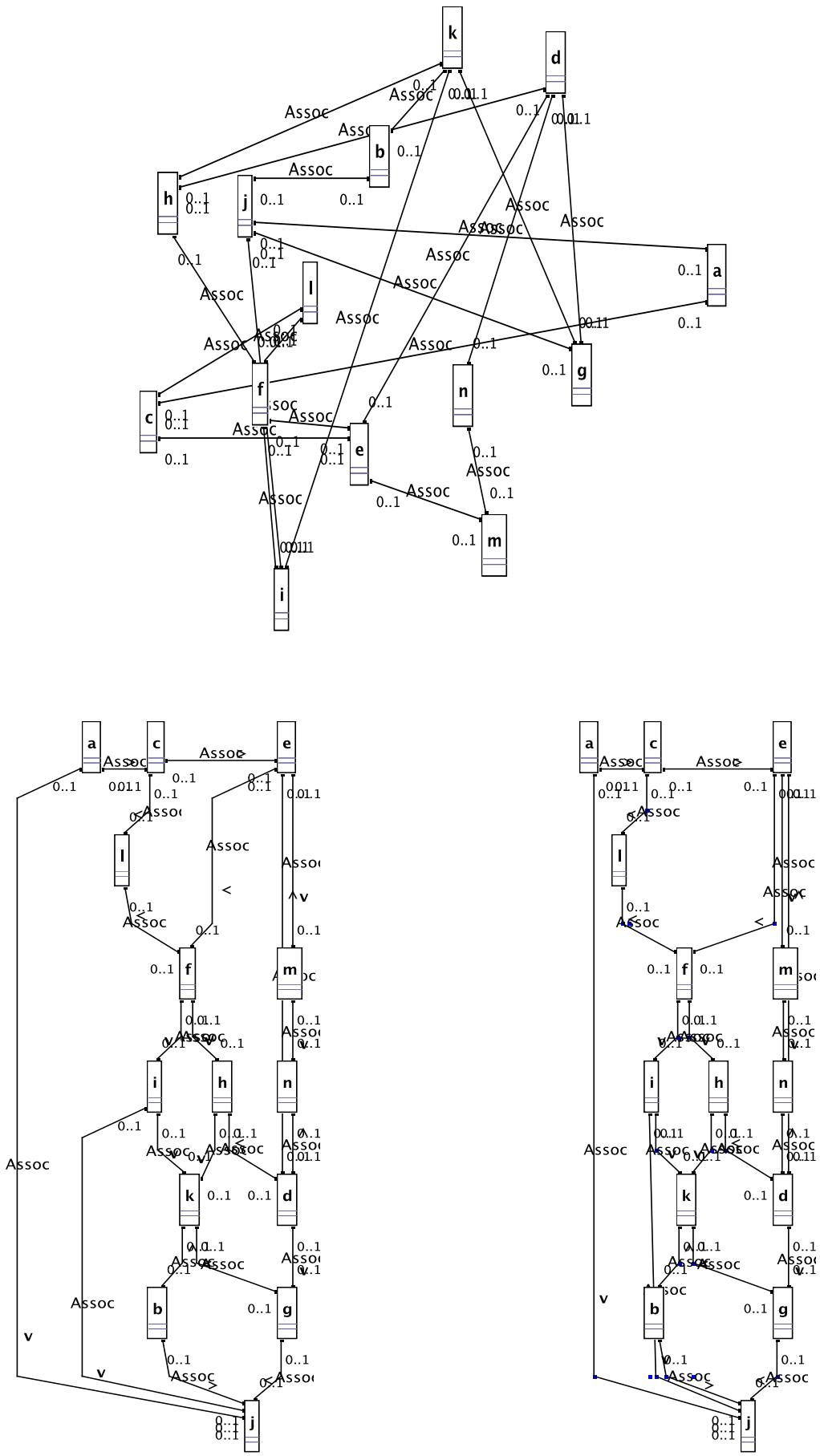


Figure 1: Two sample layouts for an artificial class diagram. A hand drawn layout is on the top, a basic layout is on the down left and a layout with edge bend minimization is on the down right.

2. LAYOUT ALGORITHM FOR CLASS DIAGRAMS

Implemented layout algorithm is originally based on a *C++* implementation [6] of an algorithm given by Nummenmaa [7]. The layout algorithm constructs first an abstract graph from the FUJABA objects (JPanel and JLine). Then this graph obtained from different objects and their relations is given as input to the layout algorithm. The steps that the layout algorithm performs are as follows:

1. Graph is connected and biconnected.
2. If graph is non-planar, then it is planarized by adding dummy vertices [4].
3. Graph is maximalized [11].
4. Canonical numbering for the vertices is calculated [7].
5. Visibility representation is constructed from the canonical numbering [7].
6. The new positions for vertices is found by using the visibility representation [6].
7. The total number of edge-bends is minimized by optimizing the positions of vertices and the routing of edges [6].

The new coordinates of classes are updated after the last phase. Basically, the layout algorithm works with any kind of diagrams that contains objects (=vertices) and connections between these objects (=edges).

The basic theoretical properties of the layout algorithm are the following:

1. Algorithm runs in linear time if the input graph is planar, otherwise its running time is bounded by the number of added dummy vertices in Phase 2.
2. Obtained drawing is a polyline grid drawing.
3. If graph corresponding to the class diagram to be drawn is planar, then there is no edge crossings in the layout.
4. If graph is non-planar, then there will be edge crossings in the final layout.
5. Classes do not overlap in the layout.
6. Algorithm works with variable size classes.
7. Algorithm does not change the sizes of the classes. Only the positions of classes and edges are changed.
8. It is possible to construct different layouts for the same class diagram by running the algorithm again.
9. It is possible to use a procedure that tries to minimize the number of edge bends of the layout.
10. Algorithm does not take account any semantic information concerning classes and their relations.

See Figure 1 for two sample layouts of an artificial class diagram containing 14 classes. A hand drawn class diagram is on the top and a new basic layout is on the down left. In the layout the sizes of classes do not change, only new positions are calculated. Edges are drawn with at most two bends for an edge.

An experimental layout is on the down right in Figure 1. In this layout the total number of edge bends is tried to minimize by calculating better positions for classes. The minimization of the total number of edge bends is one of our research problems. The implementation for the edge-bend minimization is still in progress.

3. IMPLEMENTATION

The algorithm is implemented with Java, since FUJABA requires that. The code is, at the moment, divided into eight files. Some files and classes are there for the FUJABA plugin (the Layouter is added as a standard plugin), but the important files considering the layout are

UMLClassDiagramAdapter.java
This file contains a class which is used to convert the FUJABA's UMLClassDiagram into a form that can be used in the layout-algorithm. It also converts the new layout back to FUJABA. The same layouter can also be used for other types of diagrams, but the implementation needs a different adapter.

NodeInfo.java and **EdgeInfo.java**
These two files are added so that the layout-algorithm is more general purpose. Diagram is first converted into NodeInfo's and EdgeInfo's and only then fed to the layout-algorithm, so these classes work as sort of a bridge. With the help of these classes the actual algorithm does not need to be changed every time it's used in a different program.

SimpleGraph.java and **LayOut.java**
These two files contain the actual implementation of the algorithm. Originally the algorithm was implemented with C++ and this version is translated from that one. New methods are also added due to differences between C++ and Java.

4. USAGE

Using the Layouter in FUJABA is simple.

1. Start FUJABA with this Layouter added as plugin.
2. Open a project.
3. Open the Class diagrams folder and choose a diagram.
4. The layout-button appears at the bottom of the vertical panel of buttons. Press the button.

Every time you press the relayout-button again, you get a different layout. So by doing this, you can try different layouts until you find one that satisfies you.

At this moment it's not possible to exploit manual changes on diagrams when using the layout algorithm multiple times.

The whole diagram is relayouted when you press the relayout-button. Also the manual changes in layout will be gone.

Currently there is no a button for the edge bend minimization. To test this experimental feature a function call in class UMLDiagramAdapter.java need to be changed.

5. FUTURE WORK

The implementation of this layout-algorithm is still very much in progress and at this moment the plugin cannot be downloaded from anywhere else but the FUJABA repository. The publishable version of our plugin should be ready December 2003.

Layout algorithm needs still some fine tuning and bug-fixing. Phases 2 and 7 of the algorithm are hard combinatorial optimization problems, and our goal is to study efficient heuristic methods for them. Since the algorithm does not take account any semantic information about the underlying class diagram, it might be possible to improve the readability of the layout using this kind of information. For example, a class that inherits another class, should be drawn below the base class. Currently algorithm does not support any additional information given by user (constraints) when constructing the layout. For example, user might want to draw a subset of classes closely together or choose a set of classes which positions in the layout are not allowed to change.

From the point of view of aesthetics, it could be possible to consider some aesthetics better than they are implemented in the current algorithm. Such aesthetics are the maximization of the smallest angle between edges and the minimization of the ratio of the sides of the smallest rectangle covering the layout. It should be noticed that often the optimization of one aesthetics contradicts with the optimization of another aesthetics.

Layout algorithm is designed for abstract graphs. Therefore it is, basically, possible to apply our algorithm for any kind of diagrams that contain objects and relations between those objects. If the considered diagram type (for example, component diagrams) contains multiple edges between objects, it is easy to modify the layout algorithm to support this feature. Since the current layout algorithm does not take account semantic information about the underlying graph, there will be difficulties to apply the algorithm for diagrams that contain additional information on the relation of objects (for example, hierarchical structure of statecharts).

To use the layout algorithm for other diagram types, it is in most cases enough to modify the adapter class (UMLClassDiagramAdapter.java) that converts the FUJABA class diagram to the graph format that is understood by the layout algorithm. If there is a need to take account any semantic information on a given diagram type, at least the adapter file and the main file of the drawing algorithm (Layout.java) need changes. This latter task, depending on the wanted drawing conventions and constraints, might be very complex.

6. REFERENCES

[1] C. Batini, L. Furlani, and E. Nardelli. What is a good diagram? In *4th Int. Conf. on the Entity Relationship Approach*, pages 312–319, 1985.

[2] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing*. Prentice Hall, 1999.

[3] Fujaba tool suite 4.0. Available at `http://http://www.uni-paderborn.de/cs/fujaba/`.

[4] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. ACM*, 21:549–568, 1974.

[5] C. Kosak, J. Marks, and S. Shieber. Automating the layout of network diagrams with specified visual organization. *IEEE Trans. Syst. Man Cybern.*, SMC-24(2):440–454, 1994.

[6] T. Männistö, T. Systä, and J. Tuomi. SCED report and user manual. Technical Report A-1994-5, University of Tampere, Dept. Comp. Sciences, 1994.

[7] J. Nummenmaa. Constructing compact rectilinear planar layouts using canonical representation of planar graphs. *Theor. Comp. Sci.*, 99:213–220, 1992.

[8] T. Poranen, E. Mäkinen, and J. Nummenmaa. How to draw a sequence diagram. In *Proc. 8th Symp. on Programming Languages and Software Tools*, pages 91–102, 2003.

[9] H. Purchase, J.-A. Allder, and D. Carrington. Graph layout aesthetics in UML diagrams: user preferences. *J. Graph Alg. Appl*, 6(3):255–279, 2002.

[10] H. Purchase, R. Cohen, and M. James. Validating graph drawing aesthetics. In *Proc. Int. Symp. on Graph Drawing*, number 1027 in LNCS, pages 435–446, 1996.

[11] R. Read. A new method for drawing a planar graph given the cyclic order of the edges at each vertex. *Congr. Numer.*, 56:31–44, 1987.

[12] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[13] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, SMC-18(1):61–79, 1988.

A Database Schema Diagram Plugin for Fujaba*

Ari Seppi
Department of Computer Sciences
University of Tampere
Kanslerinrinne1, FIN-33014, University of Tampere, Finland
ari.seppi@uta.fi

Jyrki Nummenmaa
Department of Computer Sciences
University of Tampere
Kanslerinrinne1, FIN-33014, University of Tampere, Finland
jyrki.nummenmaa@cs.uta.fi

ABSTRACT
Typically present-day software uses relational databases to store persistent data. We describe a way to build an object-oriented interface to a relational database. It is possible to either edit the relational database schema graphically or to reverse engineer the relational database description from the database. It is possible to automatically generate an object-oriented interface for the relational database. We have implemented our methods as a plugin for the Fujaba platform. We also describe shortly some experiences of using the plugin.

Categories and Subject Descriptors
H.2.1 [**Database Management**]: Logical Design; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms
FUJABA plugin, database schema diagrams

1. INTRODUCTION
Most present-day information systems use relational databases to store the persistent information[1]. Object-oriented software systems are no exception in this respect. There is a natural need to reflect this situation also in the software development tools.

The basic requirement for this is that the software development tools are able to model both object-oriented software and a relational database. The present-day standard for OO modeling is the use of UML. A relational database should be modeled as it is, ie. tables with attributes, along with primary and foreign keys.

These models should be used in the software development toolset to create an object-oriented interface to the data in the relational database. It should be understood that implementing this interface is a boring and error-prone task, and requires re-engineering whenever the database schema changes. Therefore, it is best to automate this activity. It should also be understood that just creating an interface to the relations is not enough. It is natural that views and queries are needed as the system is being developed. It should also be possible to model those as a part of the relational model and to create object-oriented access to the views and queries as well.

We describe a way to accomplish these aforementioned tasks within the Fujaba [2] framework. We have implemented a Fujaba plugin to make these tasks possible in practice.

We assume that the reader is familiar with relation database basics, as well as object-oriented software development.

2. THE RELATIONAL LEVEL
Graphically, the relational database is represented as a graph, where the set of nodes is the set of relations and the set of edges is the set of foreign-key relationships between the relations. An example is given in Figure 1.

However, there are other important elements in the relational level, which also should be modeled. Although it is possible to query single relations, it is likely that many or most queries will in practice access data in several relations. For software development, these queries are equally important to model.

We have chosen two ways to model queries. First of all, it is possible to model a view by selecting the attributes to be included in the resulting view and by giving the textual representation of the condition for the query representing the view. Graphically, this is like making a new relation based on the existing relations.

Secondly, it is possible to define a query, which uses a number of relations. The main difference between a view and a query is as follows. When the view is used to access the database, the result is a set of rows from the (virtual) view relation. However, when the query is used to access the database, each row in the result in fact consists of the full rows of the relations used to define the content of the query. That is, the answer contains full rows of original database relations, which can then be used to e.g. update the data in the database (unlike the rows of a view, which may, on the

*Supported by the Academy of Finland (Project 51528)

other hand, be more appropriate to represent the required answer).

The primary working method in mind with this toolset is that the designer describes the relational database and views and queries. These are then used to generate object-oriented software, as described in the next sections. We have some specific ideas on how to create this object-oriented interface to support flexibly application development.

However, sometimes the database is changed without updating the relational database diagram or it already exists. In these cases it is necessary to reverse engineer the database description. We acknoledge the existence of the REDDMOM toolset to do this reverse engineering. [5]

Due to shortage of resources, it seemed like an easier choice to implement straightforward reverse engineering based on the description of the database as SQL files or data definition statements in the database using JDBC access. If more resources are available on a later date, then maybe the more complete REDDMOM reverse engineering can be used. However, this implies a need to unify the internal data structures, to match the structures used by our forward engineering software.

The present reverse engineering simply identifies the tables and their attributes along with their data types from the table descriptions. Similarly, the primary and foreign keys and unique definitions are just taken in a straightforward manner from the table definitions. It is also possbile to guess foreign keys based on attribute names. Foreign key guessing uses a very simple algorithm: First it checks if two tables have one or more attributes with same names. After that it accepts for foreign key guessing the attribute groups that form a primary key or unique constraint.

The tools created by us do not use any real data access in the reverse engineering, and apparently a more complete reverse engineering tool such as the REDDMOM might prove useful.

We have no reverse engineering support for program source code. However, it has been found that concentrating on the data is more efficient at first phase (e.g. [3]). Of course, this does not mean that source code reverse engineering would be useless, but it does support our choice to prioritize reverse engineering of data.

Graphically, the drawing area figures for the relations are as follows. Tables are rectangles and views dashed rectangles. Foreign key relations between tables are shown as arrows pointing from foreign key's table to table where key's attributes form a primary key. Query relations are displayed as lines without an adornment on either end and finally a view's relation to a table is similar to composition in class diagrams.

It is possible to output the SQL statements to create the database and define the views. A database description in HTML form is also produced.

3. THE OBJECT-ORIENTED INTERFACE TO THE DATABASE

The automatic creation for an object-oriented interface to a relational database naturally requires some design principles. That is, there must be some rules by which the respective classes are formed.

Our software is built with primarily JDBC access in mind. JDBC, of course, provides a lot of flexibility as it can be used to access databases from different vendors.

To start from the basic choices, there must be some data types for the relation attributes. The data types used by default in JDBC are, however, quite poor for a number of reasons. First of all, one should not use primitive data types, as they do not enable representation of null values. Even if we get around this by using wrapper classes such as Integer instead of int, we still face the situation that the services given by different data types are different. There are no similar constructors or setters from string values and there is no consistent error management, for instance. Therefore, we chose to implement our own SQL data types to wrap the data in a unified wrapping primarily to better support string operations, JDBC access, and error management.

Based on our SQL data types, a class is formed for each relation and view. These classes contain SQL data type attributes for table attributes, and methods to set/get values (also as strings) as well as methods to access the database for insert, modify, delete and retrieval of a single row.

In fact, two classes for a relation R are initially created: a database class DB_R to hold all automatically generated code for relation R, and a class R, which inherits DB_R. The user is to add all necessary additions to class R, as DB_R may be regenerated later with minimal implications to existing software.

In addition to the above methods, the class R also includes a method to make a query of the form SELECT * FROM R WHERE w, where w is a SQL condition to be given as runtime input for the method. For each view, similar classes are created to access the data through the view.

The treatment of queries is not similar to the relations and views, as one may guess from the discussion in the previous section. The difference between views and queries is in their object-oriented interface as follows. Assume that a query Q is of the form SELECT * FROM R1,...,Rn WHERE w, where w is, again, an applicable condition. Then, the class Q generated for Q contains methods for making query Q. However, when the results are traversed, the values for each row are returned as objects for all classes R1,...,Rn, instead of just one class containing all attributes of the query. This way, one gets objects which can be used to update the data in these relations directly. Figure 2 contains classes generated from the relational database schema in Figure 1.

The plugin can be used to create the Java code for the classes described above. Fujaba's reverse engineering facilities may then be used to add these classes to the class diagrams.

4. IMPLEMENTATION

The current implementation of the plugin has the functionalities described above.

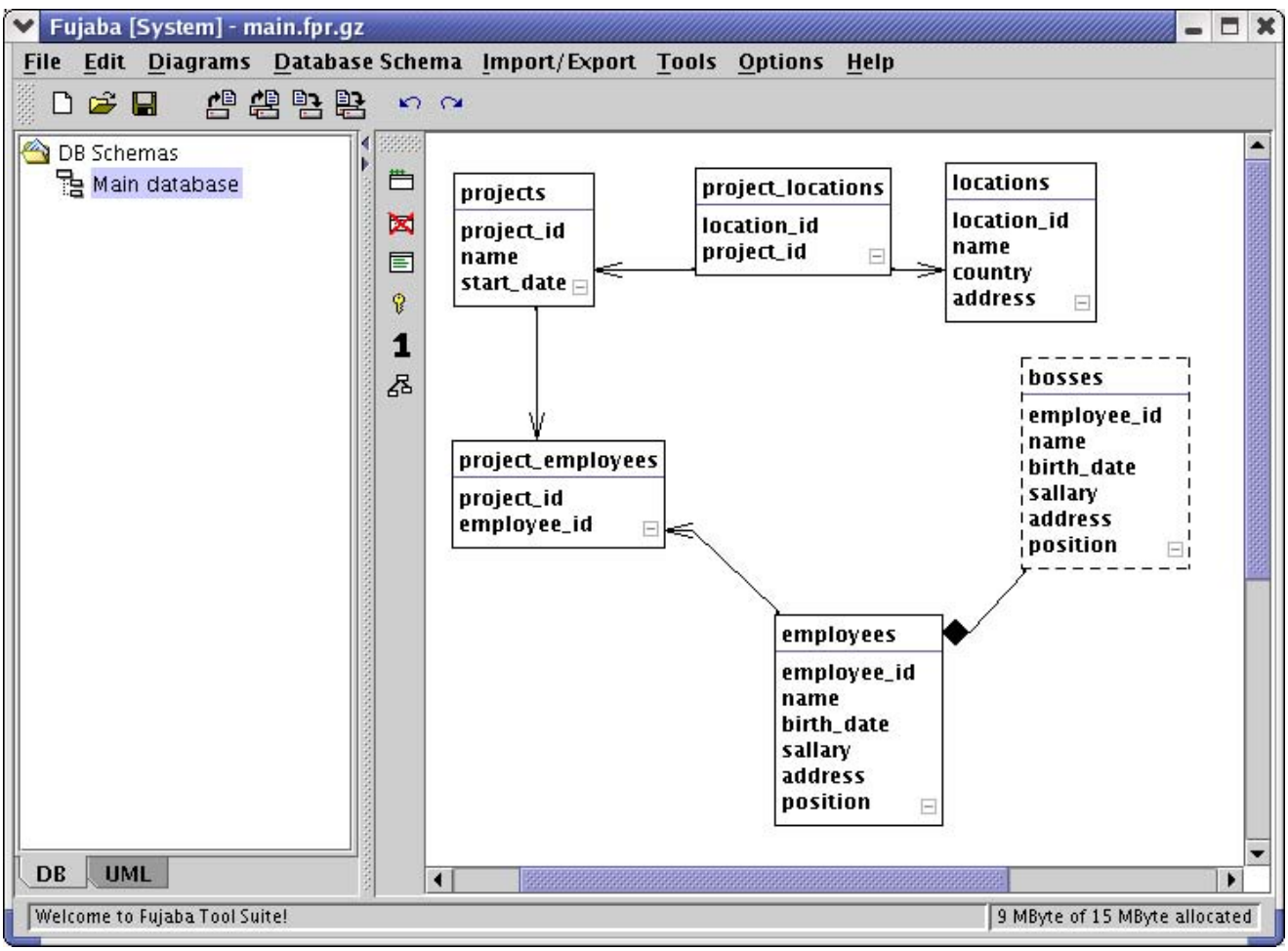


Figure 1: An example relational database schema

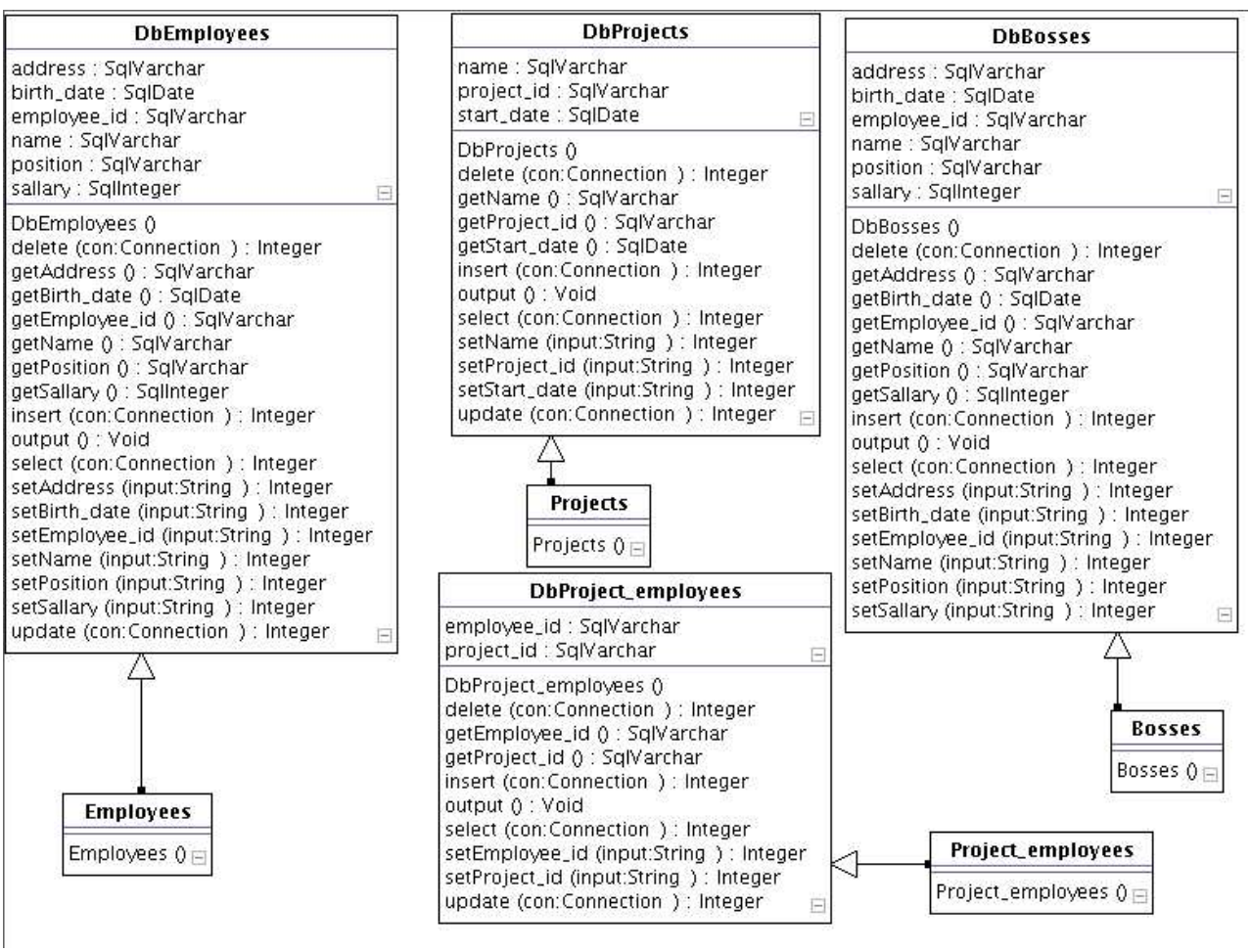


Figure 2: An example class diagram computed from the relational diagram in Figure 1

The first phase in a plugin design is the object modelling: how to model the schema objects and their relationships.

First object class that comes to mind is naturally a database schema itself, the second is a database table. After them we move on to attributes and notice that they are complicated enough structures to be objects of their own too. The same goes for unique and foreign key constraints. Unique constraint doesn't seem to need its own class (a list of names would do), but implementing separate class is clearer and - in this plugin's case - also necessary because of Fujaba's structures. Two more classes are needed: one for views and one for queries.

Secondly the objects in the classes need relations to each other. Attributes and unique and foreign constraints are related to a table (foreign constraints to two tables), and because of Fujaba's drawing model, the foreign keys also have to be referred to in a schema so that they can be drawn on the drawing area. Foreign keys and unique specifications also refer to attributes.

In addition to actual scheme objects there was a need for helper classes to use Fujaba's drawing functionality, more accurately, the helper classes were needed to create the grabbing dots between a table and a line connecting it with other table.

Because some of the GUI objects in Fujaba core worked only with UML object types, they could not be used directly on the plugin. That is why the plugin contains classes that are largely identical to classes in the core.

Unfortunately, the class diagram of the plugin is too large to be given here. However, the implementation of the plugin was not an unreasonable task using the guidelines of the Fujaba project.

5. EXPERIENCE

The software methodology described here as evolved from a set of command-line tools, where the relations, attributes and such were originally specified using text files. The command-line tools were then used for generating SQL and Java code. We call this toolset *Dbswtool*. These tools were then implemented in Fujaba in the form described above. The tools and our software development methodology has up to this point been used to develop three real applications. All these applications use, in fact, Java servlets as the main technology for the user interface.

The first of these applications is a group calendar, which has fairly small number of only nine relations. These were specified using Dbswtool. However, maintaining the design data was maybe quite tedious and the generated code did not originally contain all the useful features now present. As a consequence, the generated object-oriented interface to the database was not used maximally in the application development.

The second of these applications is a set of simple course evaluation forms. Initially, there were five relations, but they were later unified into just three. The Fujaba implementation was used and found easy, along with the rearrangement of the database. The database access has been completely based on the generated interface.

The third application is a teaching management system. This is by far the largest of these systems, with a database of currently 48 tables. The development of the system is expected to be more or less over in a few months. The Fujaba implementation is in use. There will be a fairly high number of queries and views in the system, once it is completed.

However, with the database designed and basic servlet technology implemented, software development is found to be relatively easy using our software development methodology and the implemented Fujaba toolset.

6. FUTURE WORK

We believe that using the toolset now will give important feedback on its suitability to application development. There are some issues to be considered in the future development of the toolset.

It might be beneficial to be able to access the REDDMOM reverse engineering and evolving database management toolset. However, this requires a careful study of the internals of both toolsets, ours and REDDMOM.

Secondly, the database design is now only done on the relational level. There is a natural need to unify this into the object-oriented modeling. One possible approach would be to follow the database design ideas of Blaha et al. [4]

Thirdly, it is worth investigating whether something more would be needed to generate from the database definitions. Practical use of our toolset implementation will help to evaluate this.

7. REFERENCES

[1] R. Elmasri and S. Navathe. *Fundamentals of Database Systems 3 ed.* Addison-Wesley, 2000.

[2] Fujaba tool suite 4.0. Available at `http://www.uni-paderborn.de/cs/fujaba/`.

[3] J. Henrard, J.-L. Hainaut, J.-M. Hick, D. Roland, and V. Englebert. Data structure extraction in database reverse engineering. In *Advances in Conceptual Modeling: ER '99 Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling, Paris, France, November 15-18, 1999, Proceedings*, pages 149–160, 1999.

[4] W. P. M.R. Blaha and J. E. J.E. Rumbaugh. Relational database design using an object-oriented methodology. *Communications of the ACM*, 31(4):414–427, 1988.

[5] Reengineering of distributed (federated)databases for multimedia objectoriented middleware. Available at `http://www.upb.de/cs/reddmom/`.

Turning FUJABA into a Collaborative Tool*

YC 'Vik' Nuckchady
Department of Computer Sciences
University of Tampere
Kanslerinrinne 1, FIN-33014, University of Tampere Finland
vik@cs.uta.fi

ABSTRACT

This paper describes a plug-in *ColFuj* that turns FUJABA into a collaborative tool. The collaborative environment on which it is based is introduced and a explanation is given on how FUJABA actions are made distributive over a network of collaborators.

General Terms

FUJABA plug-in[2], collaborative environment, object wrapping

1. INTRODUCTION

It is said[1] that the Internet has been invented so that scientists could collaborate. However, the current state of collaboration is passive. That is colloraborators on a project make use of assistive technologies such as email and instant messaging to communicate ideas and decisions. Indeed, there are very few applications (file-sharing programs, multiplayer games) that allow users to manipulate collectively shared data in real-time. In this paper, we propose an architecture and an environment that can be used with FUJABA in order to make it a collaborative application.

The collaborative architecture is based on a centralized network, with Participants (FUJABA clients) connected to a Coordinator as shown below Fig. 1. This particular arrangement evolves over the course of a session as the number of active Collaborators change over that session. The basic star network can break into two or more star networks or join others to form a larger one with the aim of improving the performance and efficiency of the session[3].

2. DESIGN AND IMPLEMENTATION

The plug-in is made up of two parts, a Collaborative Core[1] and a set of Command Objects.

*Supported by the Academy of Finland (Project 51528)
[1]Research in progress at the University of Tampere, Finland

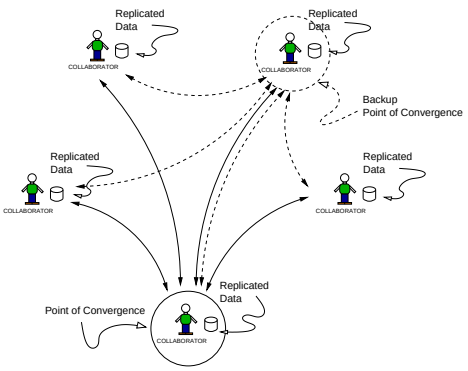


Figure 1: Network arrangement of Participants and Coordinator in a session.

The Collaborative Core is made up of two components, the Coordinator and the Participant. The latter is used by a stand-alone application, for example FUJABA, to communicate data to the Coordinator. The latter runs independently from the Participant though it can be triggered at any participanting site. This feature is used whenever the network can be optimized for performance by rewiring the Participants to new Coordinators. The Coordinator runs three Services that Participant must connect to using the appropriate Sevice Connectors. The Services that runs are the Admin, Data and Visualization (Viz). Only the latter is an optional connection from the point of view of the Participant. The Admin Service is mainly responsible for gathering statistics data through the network of collaborators. The Data Service is used for distributing *authorized* actions on the shared data to the Participants and Backup Coordinators. The last service is used mainly for distributing User Interface actions to Participants in order to establish a sence of presence in the session. Fig. 2 below illustrates how the Model View Controller design of FUJABA is enhanced through the use of the Collaborative Core. The latter is provided in the plug-in as the jar file *CollaborativeCore.jar*.

A FUJABA object is wrapped into a Command Object so that it can be distributed and executed at the participating sites. The concept is simillar to that of Remote Procedure Calls. Indeed, in the case of distributing the effect of dragging a class on the canvas, the `setX()` and `setY()` calls to the local object is packed by the appropriate Command Object and sent to the Coordinator. The latter then broadcasts it to all the connected Participants which then have the com-

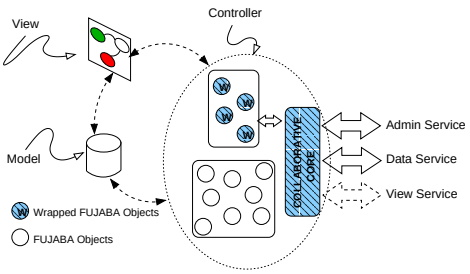


Figure 2: Decomposition of FUJABA according to the Collaborative MVC pattern. The pattern shaded components in the Controller are the constituents of *ColFuj*.

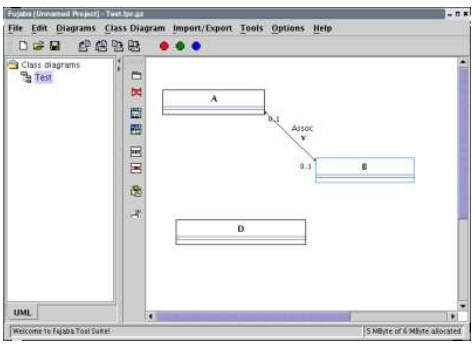


Figure 3: Screenshot of FUJABA with the plug-in ColFuj loaded.

plementary Command Object sets the `X` and `Y` values of the appropriate FSAObject to the new values.

3. USAGE

When a collaborator starts FUJABA with *ColFuj*, a small panel of three *leds* appear on the main tool bar (Fig. 3). These *leds* flicker whenever data is read from the network. Each of these led is attached to a Service Connector. So, the red, green and blue ones correspond to the Admin, Data and Visualization Service Connectors respectively. The participation of a collaborator can be toggled *on* and *off* by clicking on the appropriate led. Hence, clicking on the *red* one causes the FUJABA client to connect to and disconnect from a Coordinator. It must be noted that disconnecting from the network causes all the *leds* to turn off and turning it on activates all the *leds*. If the blue '*led* is turned off, collaboration still continues but the client will not be informed of the presence of the other collaborators. When the green *led* is turned on, the session is *frozen* and the Coordinator sends an up-to-date copy of the shared data to the newly joined Participant (FUJABA client). For that duration, the actions of the other Participants are buffered and executed after the synchronization process has completed. In other terms, changes to the data are not reflected *immediately*.

4. FUTURE WORK

The Collaborative Core mentioned above is still evolving and hence *ColFuj* will change accordingly. However, objects that have already been wrapped will still work on future versions of the Core. Currently, work is being done in extending more FUJABA functionalities from a single user mode to a collaborative one. The current *stable* version of the plug-in is that changes in the geometry of class diagrams are reflected on all FUJABA clients.

5. REFERENCES

[1] J. Udell. Internet groupware for scientific collaboration. Available at `http://udell.roninhouse.com/GroupwareReport.html`.

[2] L. Wendehals. 10 steps to buils a fujaba plug-in. Available from Software Engineering Group of the University of Paderborn.

[3] N. YC and J. Nummenmaa. An architecture for building collaborative tools in java. In *Proc. 8th Symposium on Programming Languages and Software Tools*, pages 174–186, 2003.

10 Steps to build a Fujaba Plug-in

Lothar Wendehals
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
lowende@upb.de

October 14th, 2003

Abstract

Current initiatives in the field of integrated development environment (IDE) and CASE tool integration such as Eclipse and Together indicate that tool integration has become an important issue for the IT industry. However, current integration platforms fall short to address the underlying problems of overlapping meta-models and their consistency when it comes to tool integration. Within the Fujaba Tool Suite in contrast a framework has been developed which enables an integration of tools not only at the feature and user interface level but also at the meta-model level. This tutorial enables the participant to build a Fujaba plug-in in ten steps. The participant will learn how to define a meta-model that can be connected to Fujaba's meta-model, how to extend the user interface of Fujaba and how to deploy a plug-in.

10 Steps To Build a Fujaba Plug-in

Lothar Wendehals

University of Paderborn
Software Engineering Group

Example: Diagram with Nodes and Edges

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

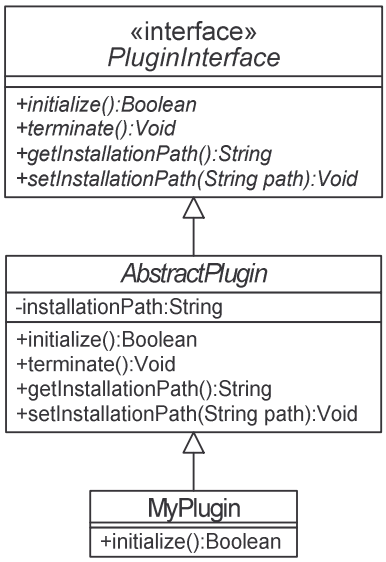
- Create a plug-in that adds a new diagram kind with nodes and edges
- Nodes should be connected to UMLClass in Fujaba's meta-model
- User can create new diagram and can add nodes and edges to it
- Fujaba should be able to save and load the diagram
- User can define shape of nodes

Step 1: Implement the Plug-in Interface

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

Two possibilities:

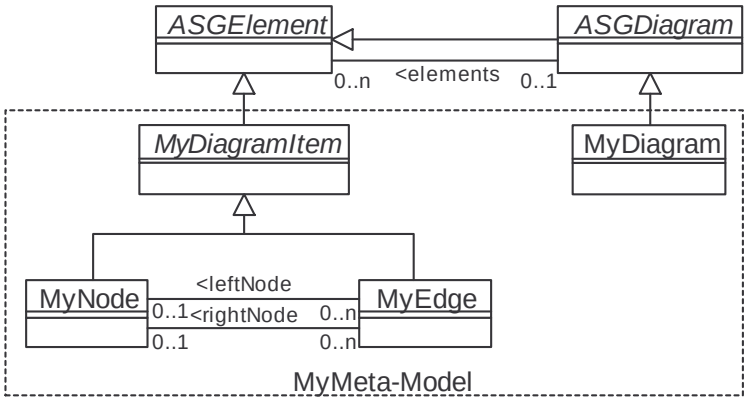
- Implement the interface de.upb.lib.plugins.PluginInterface
- Extend abstract class de.upb.lib.plugins.AbstractPlugin



Step 2: Define Your Meta-Model

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

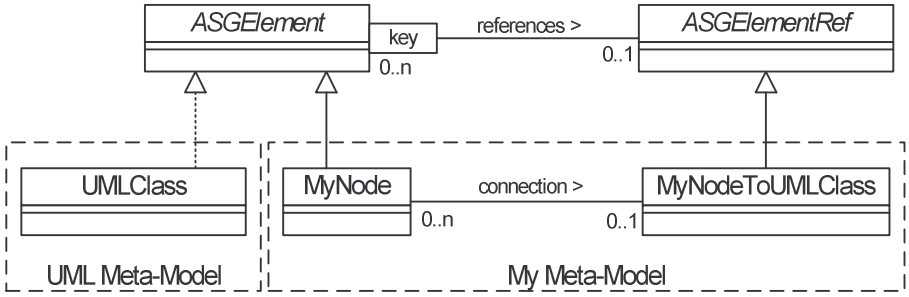
- Build the new meta-model with Fujaba and generate code
- Use the Abstract Syntax Graph (ASG) classes as super-classes
- Saving and loading is managed by ASGElement



Step 3: Connect to Fujaba's Meta-Model

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Connect MyNode to Fujaba's UMLClass by using the Meta-Model Integration Pattern
- An bi-directional association is established between MyNode and UMLClass
- Fujaba is still compilable without the plug-in



Step 4: Implement Actions

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

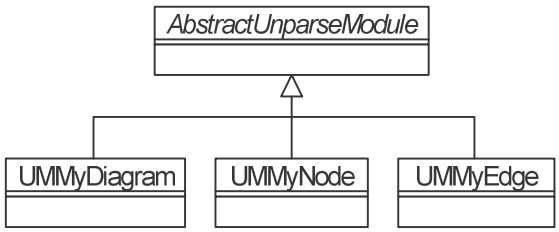
- Implement actions for creating MyDiagram, MyNode and MyEdge

```
public class NewMyDiagramAction extends AbstractAction
{
  public void actionPerformed (ActionEvent event)
  {
      // create diagram and add to project
      MyDiagram myDiagram = new MyDiagram ();
      UMLProject.get().addToDiags (myDiagram);

      // show diagram
      FrameMain.get().createNewTreeItems();
      FrameMain.get().selectTreeItem (myDiagram);
  }
}
```

Step 5: Visualize the Meta-Model

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

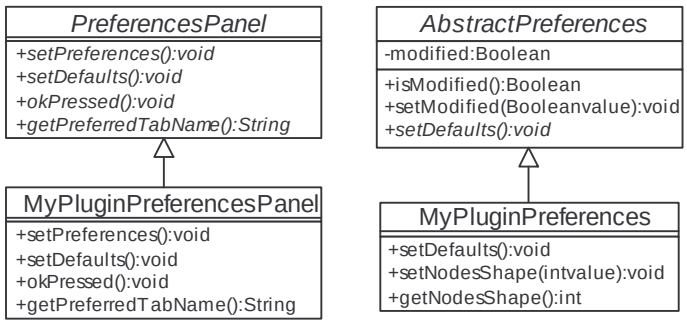
- Implement a rendering class for each meta-model class by extending de.uni_paderborn.fujaba.fsa.unparse.AbstractUnparseModule
- UnparseModules describe how elements are displayed



Step 6: Enable Configuration of Plug-in

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Implement panel for configuring options of plug-in by extending de.uni_paderborn.fujaba.app.OptionsPanel
- Panel is displayed in environment dialog for plug-ins
- Implement data storage class by extending de.uni_paderborn.fujaba.basic.AbstractOptions
- Data is saved in separate file for each plug-in by AbstractOptions



Step 7: Define the User Interface

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Create file stable.xml with description of user interface extensions
- Define for each action class the action's name, it's icon, etc.

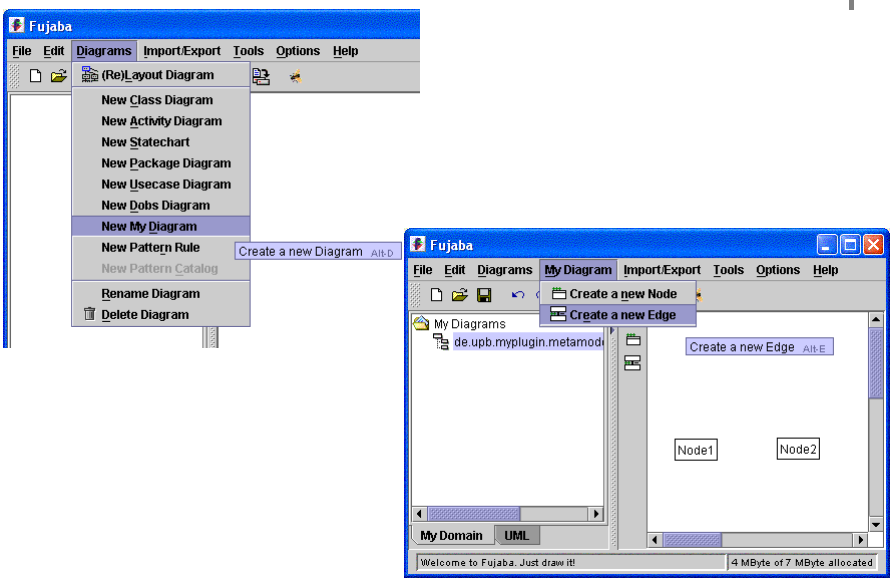
```
<Action id="newMyNode" class="de.upb.myplugin.actions.NewMyNodeAction"
  enabled="true">
   <Name>Create a new node</Name>
   <Mnemonic>n</Mnemonic>
   <ToolTip>Create a new node and add it to diagram</ToolTip>
   <Icon>de/upb/mydiagram/images/newNode.gif</Icon>
 </Action>
```

- Define menus, popup-menus and toolbars

```
<PopupMenu class="de.upb.myplugin.metamodel.MyDiagram">
   <MenuSection id="editSection">
     <MenuItem actionId="newMyNode"/>
   </MenuSection>
</PopupMenu>
```


Step 7: Define the User Interface

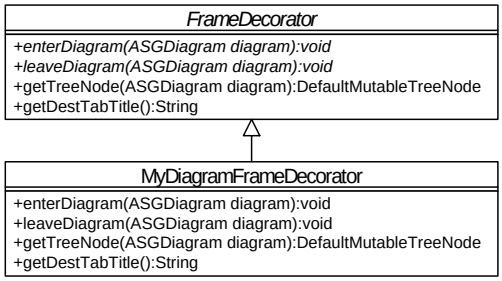
University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer



Step 8: Define the Change of the UI

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Define how the UI will look like, if MyDiagram is displayed
- Extend the abstract class de.uni_paderborn.fujaba.app.FrameDecorator
  - Define changing of menus, toolbars, etc.
  - Define the tree node of MyDiagram within the diagrams tree
- Add new FrameDecorator to list of FrameDecorators during initializing the plug-in



Step 9: Define Plug-in Description

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Description contains:
  - Name of the class implementing Fujaba's plug-in interface
  - Name of the plug-in
  - Version number
  - Name of the plug-in library
  - Web address where to download the plug-in
  - Required Kernel version
  - Additional paths for CLASSPATH variable
  - List of required plug-ins
  - Textual information about the plug-in with
    - Short and detailed description
    - Vendor and contact address
- Description stored in file named plugin.xml

Step 9: Plug-in Description - Example

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE Plugin SYSTEM "http://www.upb.de/cs/fujaba/DTDs/Plugin.dtd">

<Plugin pluginClass="de.upb.myplugin.MyPlugin">
  <Name>My Plugin</Name>

  <Version major="1" minor="0“ build="0"/>

  <PluginLib>MyPlugin.jar</PluginLib>

  <Source>http://www.fujaba.de/downloads/plugins/MyPlugin/1_0/MyPlugin.zip</Source>

  <Kernel major="4" minor="0" revision="1"/>

  <RequiredPlugins>
    <PluginId pluginClass="de.upb.anotherplugin.AnotherPlugin" major="1" minor="0"/>
  </RequiredPlugins>

  <Description>
    <ShortDescription>My first Fujaba Plug-in</ShortDescription>
    <DetailedDescription>This is just a test plug-in.</DetailedDescription>
    <Vendor>University of Paderborn</Vendor>
    <Contact>mailto:lowende@upb.de</Contact>
  </Description>
</Plugin>
```


Step 10: Deploy the Plug-in

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Create library with all classes, icons, etc. of the plug-in
- Create new directory within Fujaba's plug-in directory
- Copy library, plugin.xml and stable.xml to this directory
- Start Fujaba, that's it!

How to add a new diagram to Fujaba

Matthias Tichy
Software Engineering Group
University Of Paderborn

October 14th, 2003

Abstract

In the context of CASE tools, typically all applications use some meta-model for the design representation. An instance of such a meta-model resembles the design of the new application. This instance is shown as a graphical diagram to ease the comprehension for the developer of the design. For the developer of a CASE tool it imposes a burden to create the graphical diagram based on the meta-model instance and to keep it up-to-date.

The Fujaba Tool Suite provides sophisticated support for this problem in the form of the Fujaba Swing Adapter (FSA) architecture. For the graphical view the developer only needs to develop the initial mapping from the meta-model to the graphical view, whereas the later synchronization is almost automatically done by the FSA architecture.

In this tutorial we introduce the FSA architecture and its concepts. We show how it can be used for the creation of graphical diagrams. A stripped down real-life example will be used to show the pre-requisites on the meta-model side and how the initial mapping from the meta-model instance to the graphical representation is done. We finish with some tips and tricks.

The Fujaba Diagram Visualization Architecture

Matthias Tichy
Software Engineering Group
University of Paderborn

Contents

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Motivation
- Architecture
- Details
- Common pitfalls

Graphical Case Tool Design

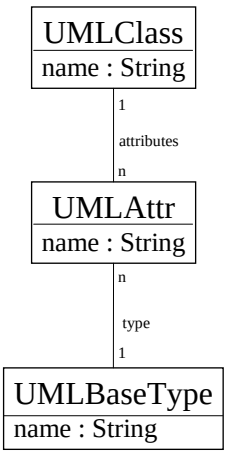
University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- graphical Diagrams are instances of certain meta-models (e.g. UML meta-model)
- A instance is changed by the user or the application itself
  - structural changes
  - value changes

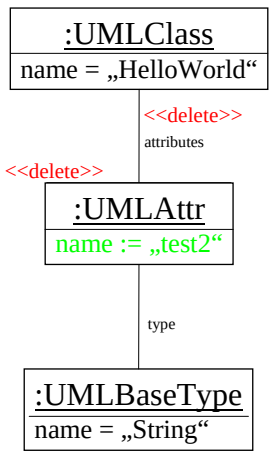
Case Tool Design

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

Meta-Model



Instance



- Structural Changes
  - deleting links
  - (deleting objects)
- value changes

Graphical Tools

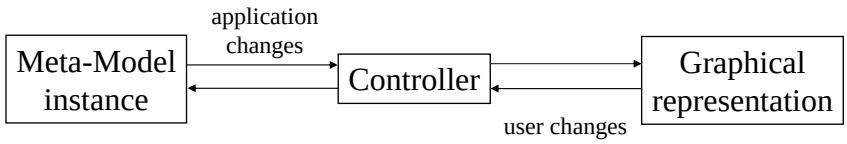
University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Graphical visualization of Meta-Model instance
- Initial creation of a graphical representation is relatively easy (the mapping)
- On each change doing the whole mapping again is not feasible
- Managing incremental differences
  - between graphical representation and model

Graphical Tools

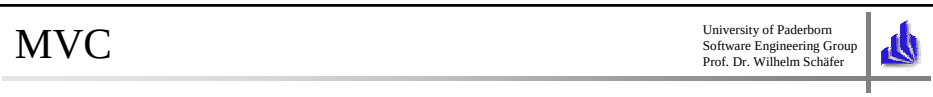
University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Problem keeping both parts in sync
  - Propagating meta-model instance changes to graphical representation and vice versa
- Solution: Model View Controller Architecture (MVC)

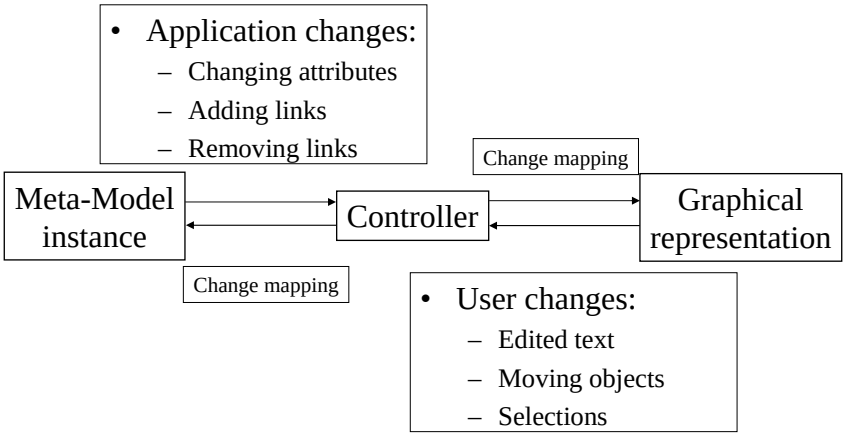


MVC

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer



- Controller forwards changes from one part to the other part



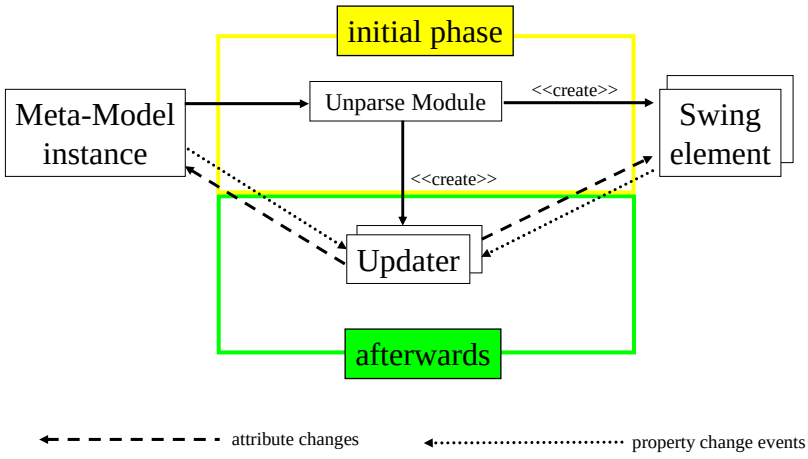
Our solution

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Swing is used for the Graphical Visualisation part
- Meta-Model supports Java PropertyChange mechanism for the change management (Observer Pattern)
  - PropertyChangeListener
  - addTo- / removeFrom- methods
  - firePropertyChange()-methods

Our solution

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

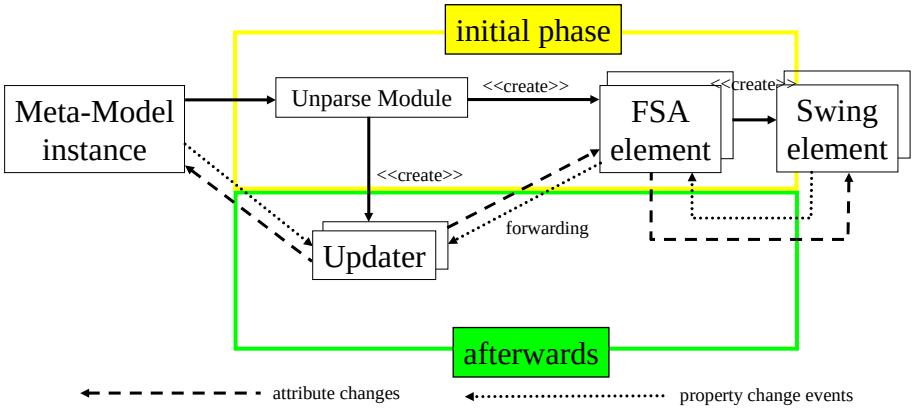


Our solution

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

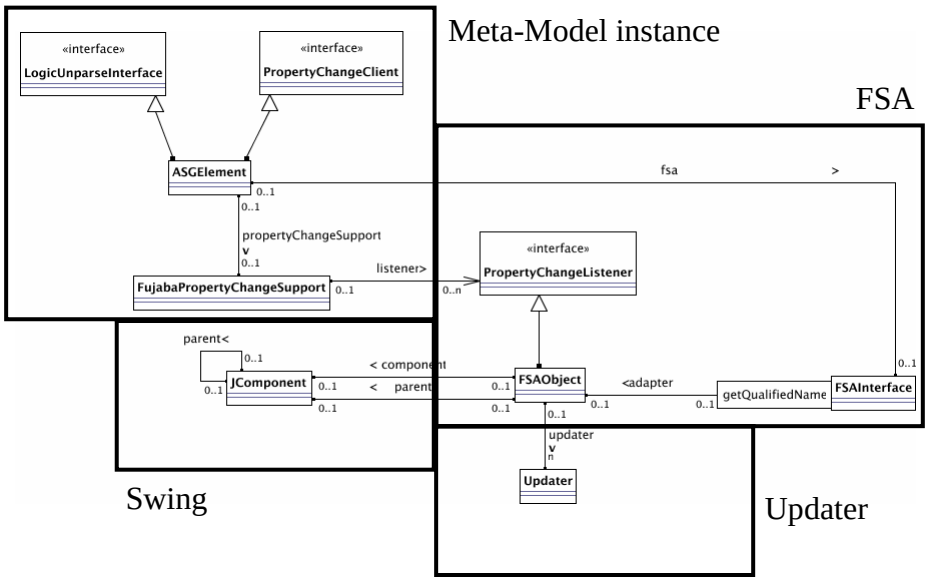
- Problem: bi-directional associations between Meta-Model instance and Swing elements needed

➢ Fujaba Swing Adapter (FSA)



Architectural Overview

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer



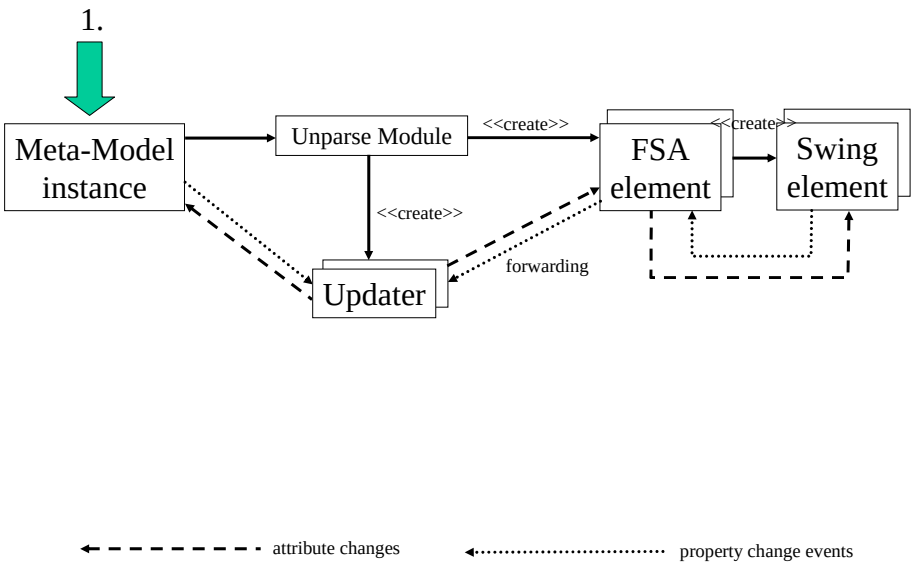
Architectural Overview

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- design issue: one logic element may be presented by multiple graphical elements
- e.g.
  - a class is shown using more than one graphical element
  - a class shown in three class diagrams
- solution: qualified assoc between logic elements and graphical elements
  - key: parentID + . + propertyName
  - this key should be **unique**

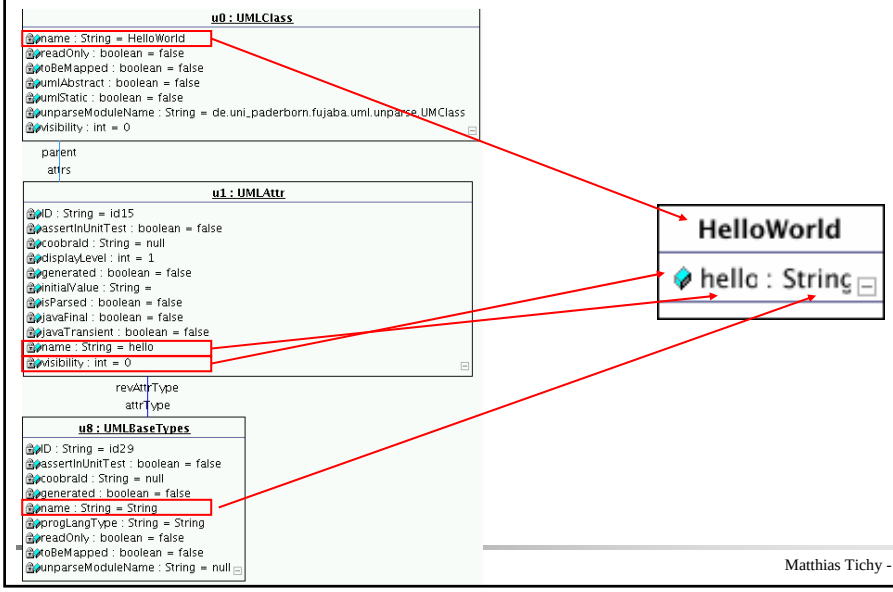
Details – next slides

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer



Example

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer



Requirements on the meta-model part

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Subclass your classes from ASGElement (or implement LogicUnparseInterface)
  - ASGElement provides support for PropertyChangeListener
    - firePropertyChange()- methods
    - addToPropertyChangeListeners()
    - removeFromPropertyChangeListeners()
- Add firePropertyChange() calls for
  - Attribute changes
  - adding and removing of links
    - eg. a new class in a class diagram
    - or a new attribute in a class

access methods / to-1 assoc

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- set-Method (eg. UMLAttr.setName()):

```
public void setName (String newName)
  {
     if (this.name == null ||
        !this.name.equals (newName))
     {
        String oldName = this.name;
        this.name = newName;
        firePropertyChange ("name", oldName, newName);
     }
  }
```

attr name

access methods / to-n assoc

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- to-n assocs are implement using collections
- Use the class CollectionChangeEvent to express adding and removing of elements
- Fujaba uses FProp*-Collection Classes
  - automatically call firePropertyChange() if you add() or delete from the collection classes

UMLClass.addToAttrs()

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

```
private FPropTreeMap attrs;
public boolean addToAttrs (UMLAttr obj)
  {
     boolean changed = false;

     if ( (obj != null) &&  (obj.getName() != null))
     {
        if (this.attrs == null)
        {
           this.attrs = new FPropTreeMap (FujabaComparator.getLessString(),
              this, "attrs");
        }
        UMLAttr oldValue = (UMLAttr) this.attrs.put (obj.getName(), obj);
        if (oldValue != obj)
        {
           if (oldValue != null)
           {
              oldValue.setParent (null);
              oldValue.removeYou();
           }
           obj.setParent (this);
           changed = true;
        }
     }
     return changed;
  }
```

assoc name

fireProperty

FPropTreeMap.put()

```
public synchronized Object put (Object key, Object value)
   {
      boolean fire = getPropertyChangeSupport() != null &&
   !this.containsKey (key);
      Object result = super.put (key, value);

      //if the key was already in this map, the PropElement fires the
   event
      if (fire)
      {
         firePropertyChange (result, value, key,
   CollectionChangeEvent.ADDED);
      }

      return result;
   }
```

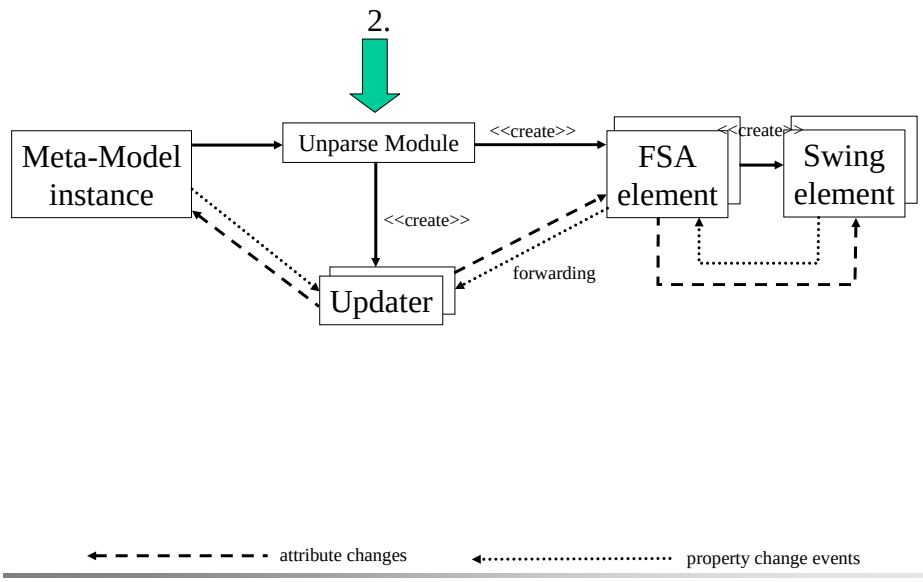
firePropertyChange call

Excursion

- Generate Meta-Model source-code including PropertyChange stuff
  - Why bother with PropertyChange stuff? Generate it!
  - Just use Fujaba! 
- add <<JAVA_BEAN>> Stereotype to every class
  - this stereotype must be created in the adding dialog
- everything is generated on the fly (needs at least upcoming Fujaba 4.0.1 or the latest Fujaba 3)

Details – next slides

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer



Unparse module

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Unparse modules build the FSA/Swing counterparts of meta-model instances.
- Each meta-model class needs one unparse module.
- Unparse modules must be named in a special way:
  - Normally: `UM + class.getName()`
  - But: UMLClass ⇒ UMClass
    - (just overwrite: `ASGElement.createUnparseModuleName()`)
  - must be in subpackage: unparse

Unparse module

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- create():
  - initial mapping to swing (resp. FSA*) classes.
- getMainFsaName():
  - name for the main entry, which is created by the unparse module.
- getChildProperties():
  - names of the children of the meta-model instance
- getContainerForProperties():
  - returns the container for the children

UnparseModule

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

```
public FSAObject create (FSAObject parent, LogicUnparseInterface incr)
  {
     UMLAttr attr = (UMLAttr) incr;
     FSAPanel mainPanel = null;

     mainPanel = new FSAPanel (incr, getMainFsaName(), parent.getJComponent());
     mainPanel.setLayout (new ColumnRowLayout (0, ColumnRowLayout.ROW));

     FSAComboBoxLabel iconLabel = new FSAComboBoxLabel (incr, "umlVisibility",
           mainPanel.getJComponent(), new UMLAttrVisibilityJComboBoxLabel());
     iconLabel.addToUpdater (iconLabel.createDefaultUpdater());

     FSATextFieldLabel nameField = new FSATextFieldLabel (incr, "name",
           mainPanel.getJComponent());
     nameField.addToUpdater (nameField.createDefaultUpdater());

     FSALabel colonLabel = new FSALabel (incr, "colon", mainPanel.getJComponent());
     colonLabel.setText (" : ");

     FSAComboBoxLabel typeLabel = new FSAComboBoxLabel (attr, "attrType",
        mainPanel.getJComponent());
     typeLabel.setModel (UMLTypeListComboBoxModel.get());
     typeLabel.addToUpdater (typeLabel.createDefaultUpdater());
     AbstractUpdater updater = new TypeUpdater (attr, "attrType",
        typeLabel.getDefaultAttrName());
     typeLabel.addToUpdater (updater);

     return mainPanel;
  }
```

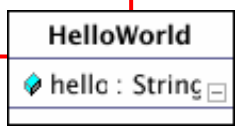
Panel

Icon

Name

:

Type



Unparse module: UMClass

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

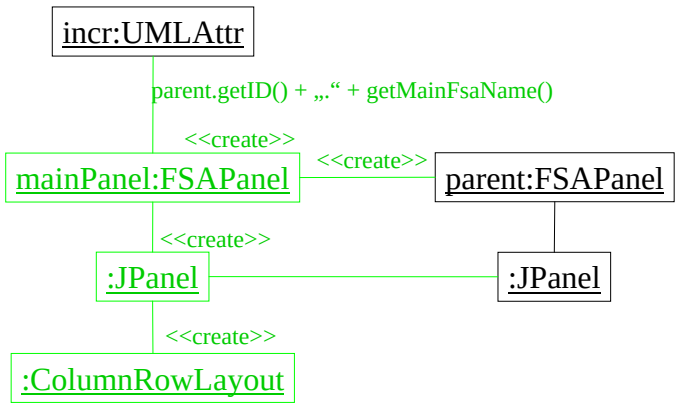
```
public void getChildProperties (Set props)
   {
      super.getChildProperties (props);
      props.add ("attrs");
}


public FSAObject create (FSAObject parent, LogicUnparseInterface incr)
   {
      UMLClass clazz = (UMLClass) incr;

      FSASeparatedPanel panel = new FSASeparatedPanel (incr, getMainFsaName(), parent.getJComponent());
      panel.setBorder (new LineBorder (Color.black));

      FSAUnderlinedObject namePanel = new FSAUnderlinedObject (incr, "classNamePanel", panel.getJComponent());
      namePanel.setLayout (new ColumnRowLayout (0, ColumnRowLayout.COLUMN));

      FSATextFieldLabel classNameTextField = new FSATextFieldLabel (incr, "name", namePanel.getJComponent());

      FSACollapsable tmpCollapsable;
      tmpCollapsable = new FSACollapsable (incr, "attributePanel", panel.getJComponent());

      return panel;
   }


public String getContainerForProperty (String property)
   {
   if ("attrs".equals (property))
      {
         return "attributePanel";
      }

      return super.getContainerForProperty (property);
   }
```


UnparseModule

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

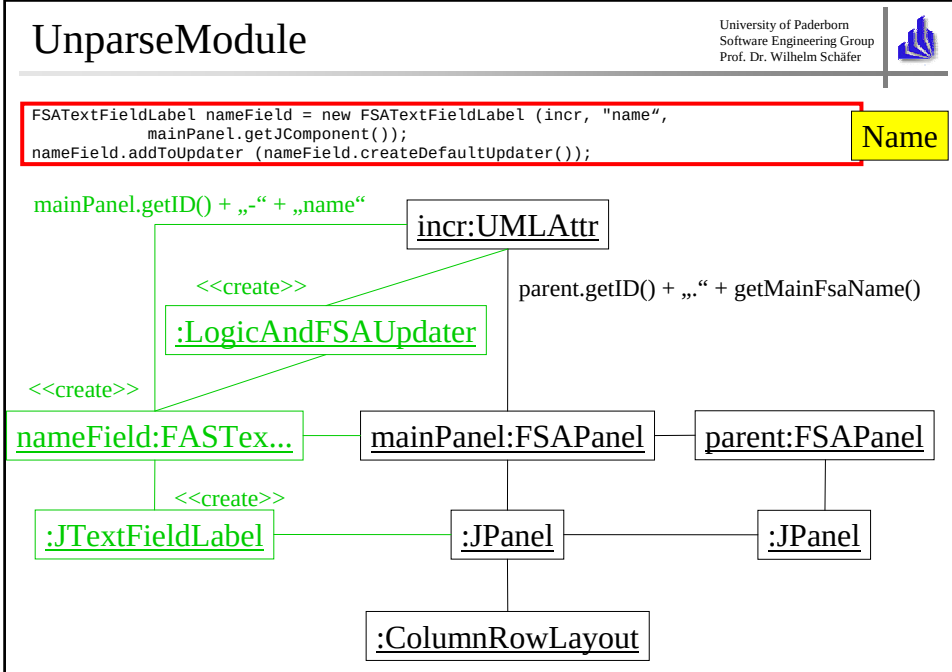
```
mainPanel = new FSAPanel (incr, getMainFsaName(), parent.getJComponent());
mainPanel.setLayout (new ColumnRowLayout (0, ColumnRowLayout.ROW));
```

Panel



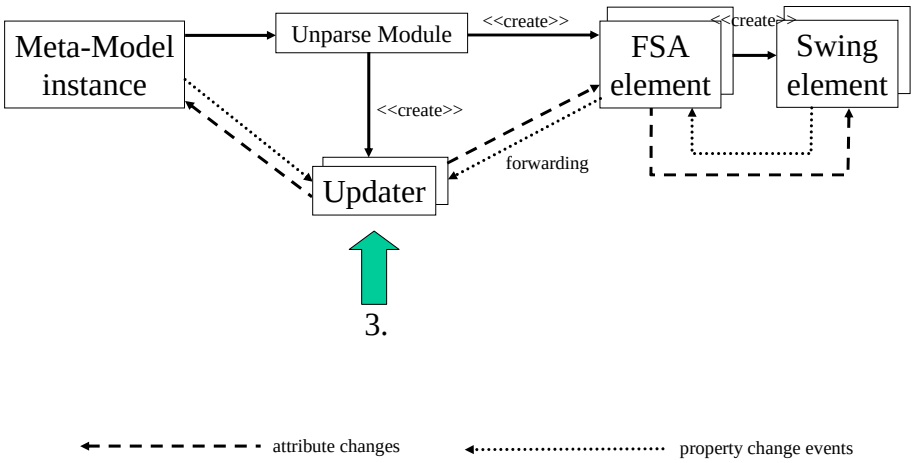
UnparseModule

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer



Details – next slides

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer



Updater and Translators

- Updater
  - forwards changes and the changed values
  - is called for each PropertyChangeEvent
- can execute arbitrary code on PropertyChangeEvents
  - e.g.: hide or show graphical elements (VisibilityUpdater)
- relevant class: AbstractUpdater
- each FSA* class has a default updater – often sufficient

Updater and Translators

- Translators
  - used for value changes
  - for example to convert types (Integer <-> String)
- Interface: Methods
  - public Object translateFsaToLogic (Object value);
  - public Object translateLogicToFsa (Object value);
- Hint: an updater is a translator, too

What FSA's are available

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Panels:
  - FSAPanel
  - FSACollapsable (collapseble panel)
  - FSAResizable (resizable panel)
  - FSALayeredPane
- Text:
  - FSALabel
  - FSATextFieldLabel
  - FSAComboBox

What FSA's are available

University of Paderborn
Software Engineering Group
Prof. Dr. Wilhelm Schäfer

- Lines:
  - FSAPolyLine
  - FSAGrabs
  - FSABends
  - FSAArrow
- Minor stuff:
  - FSACircle
  - FSASeparator

Common pitfalls

- forgotten firePropertyChange calls
- misspelled property names
- misspelled unparse module class name
- unparse module in wrong package

How to avoid FSA

- one basic unparse module for your diagram
  - create your special Swing component
- after that do anything yourself

- You may add Swing elements into FSAContainer

Thank you for your attention!

Questions?

Adapting the Fujaba Code Generation Mechanism

Susannah Moat
University College London

October 14th, 2003

Abstract

This tutorial aims to give an insight into the workings of the Fujaba code generation mechanism, and equip the listener with the necessary knowledge to adapt the system to his/her needs.

We will begin by examining the basic design and core classes involved in code generation, and then via a simple example of a university model, we will see how Fujaba progresses from a UML diagram to an internal model of the diagram, and finally to Java code. This will require us to consider how an outline class structure is generated, and then to study how association generation builds on these foundations.

We will then summarise different scenarios for adapting the mechanism and the main components of the system which would need replacing. The tutorial will finish with an opportunity for the listeners to work through a basic example of adapting the code generation mechanism, and finally discuss or query any points which may have arisen during the talk.

Adapting the Fujaba Code Generation Mechanism

Susannah Moat

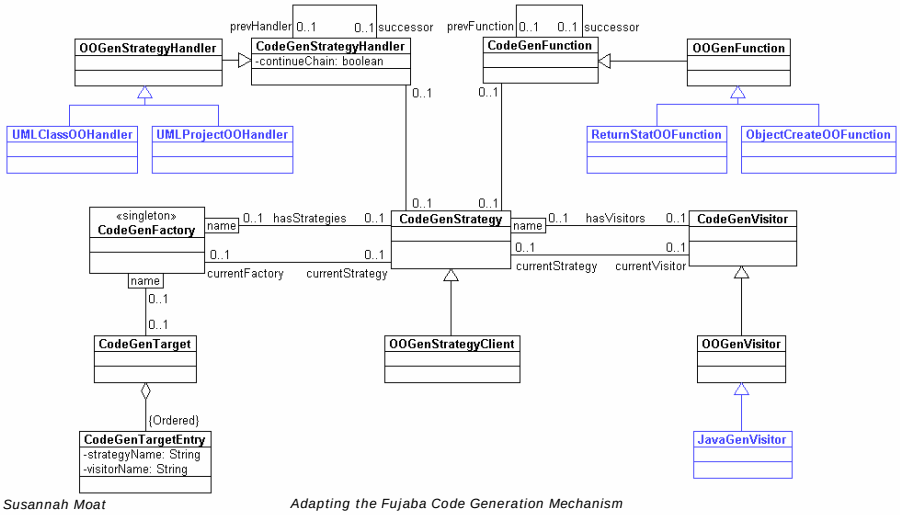
University College London

S.Moat@cs.ucl.ac.uk

Overview

- Code generation mechanism design
- Generating classes, attributes and methods
- Generating associations
- Making alterations to the mechanism
- Summary and questions

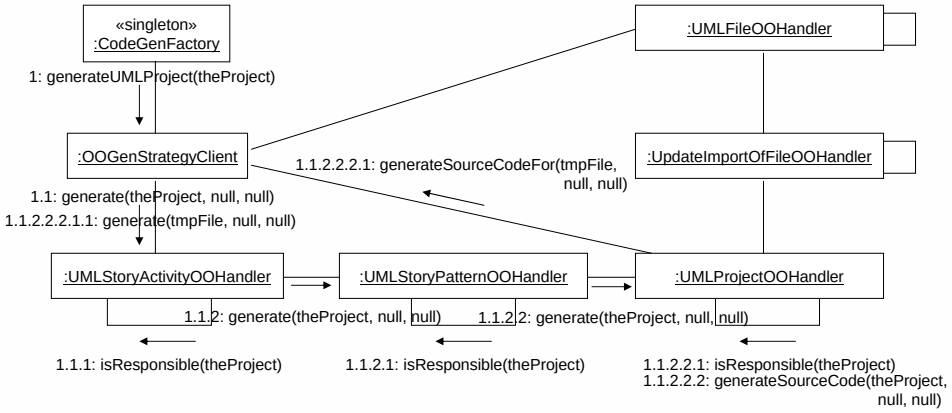
The code generation mechanism



CodeGenStrategyHandler

- Chain of Responsibility of subclasses of `CodeGenStrategyHandler`
  - Each handler subclass is responsible for a subclass of `ASGElement` (usually a `UMLIncrement`)
  - The `ASGElement` is passed along the chain until a handler determines that it is responsible for the object
  - In certain cases, the `ASGElement` is carried further along the chain to be processed by later handlers after the original handler has finished

The handler chain in action



CodeGenFunction

- Used in processing of diagrams defining behaviour, e.g. activity diagrams
- Chain of Responsibility of subclasses in same manner as `CodeGenStrategyHandler`
  - Each function subclass is responsible for generating a different type of dynamically generated statement, e.g.
    - `ReturnStatOOFunction` for return statements
    - `ObjectCreateOOFunction` for object creation statements
  - Type of code required is specified in method call on chain
  - Uses abstract description of statement – `OOStatement` etc. classes

CodeGenVisitor

- Subclasses are for specific languages
  - e.g. `JavaGenVisitor` for Java
- Contains methods for generating statements determinable from static class structure
  - Class, attribute and method declarations
  - Import and package statements
- Contains methods for generating lanaguage specific code from `OOStatement` descriptions

CodeGenFactory and CodeGenTargetParser

- `CodeGenFactory` uses a `CodeGenTargetParser` to load the desired configuration of the mechanism from an XML file
- This causes instances of the other mechanism classes to be created

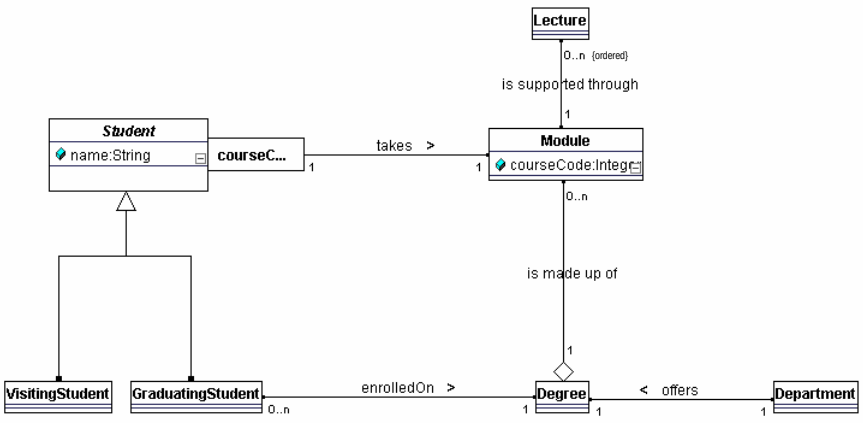
codegen/javatarget.xml

```
<codegenfactory>
 <codegenstrategy
    name="de...OOGenStrategyClient">
  <name>JavaStrategy</name>
  -- add handlers
  <object name="de...UMLProjectOOHandler"
          method="appendHandler">
  </object> ...
  -- add functions
  <object name="de...ReturnStatOOFunction"
          method="appendFunction">
  </object> ...
  -- add visitor
  <visitor name="de...JavaGenVisitor">
   <name>JavaGenVisitor</name>
  </visitor>
 </codegenstrategy>
 <codegentarget>
  <name>java</name>
  <fullName>Java (Compilable)</fullName>
  <codegentargetentry>
   <strategyName>JavaStrategy</strategyName>
   <visitorName>JavaGenVisitor</visitorName>
  </codegentargetentry>
 </codegentarget>
</codegenfactory>
```

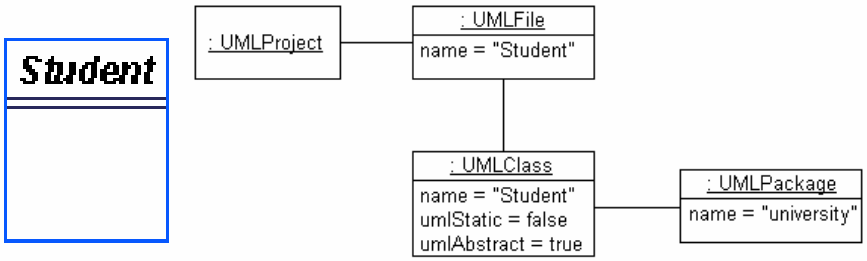
Overview

- ✓ Code generation mechanism design
- ➢ Generating classes, attributes and methods
- ■ Generating associations
- ■ Making alterations to the mechanism
- ■ Summary and questions

Example: University



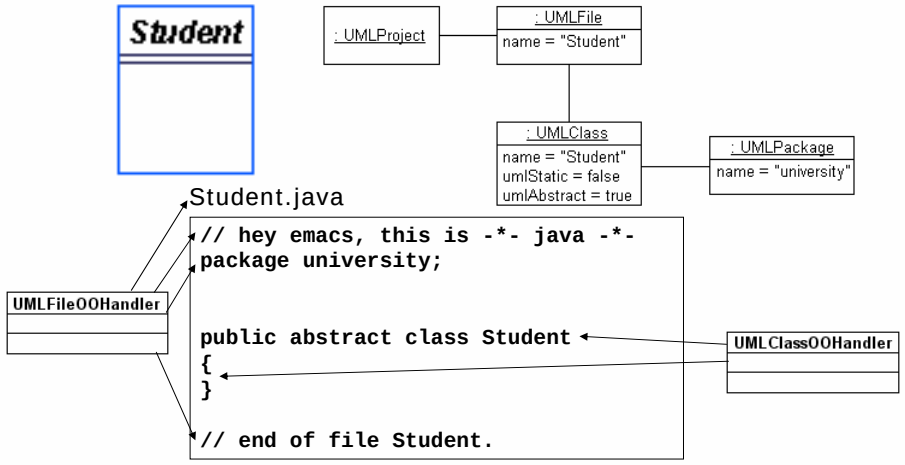
Generating classes



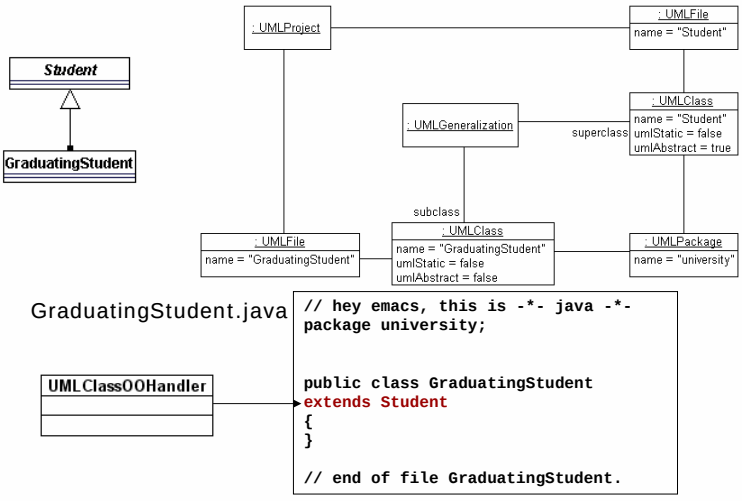
Generating classes - 2

- `UpdateImportOfFileOOHandler`
  - Checks that all necessary classes are imported
  - `UpdateImportOfFileOOHandler.continueChain == true`, so the `UMLFile` is passed along the chain to `UMLFileOOHandler`
- `UMLFileOOHandler`
  - Initialises the visitor for file creation
    - File extension determined by visitor (eg. `.java`)
  - Responsible for generating header and footer, package declaration and import statements.
- `UMLClassOOHandler`
  - Generates a class declaration via a method call on the visitor
  - Passes the `UMLAttr` and `UMLMethod` objects back to the handler chain

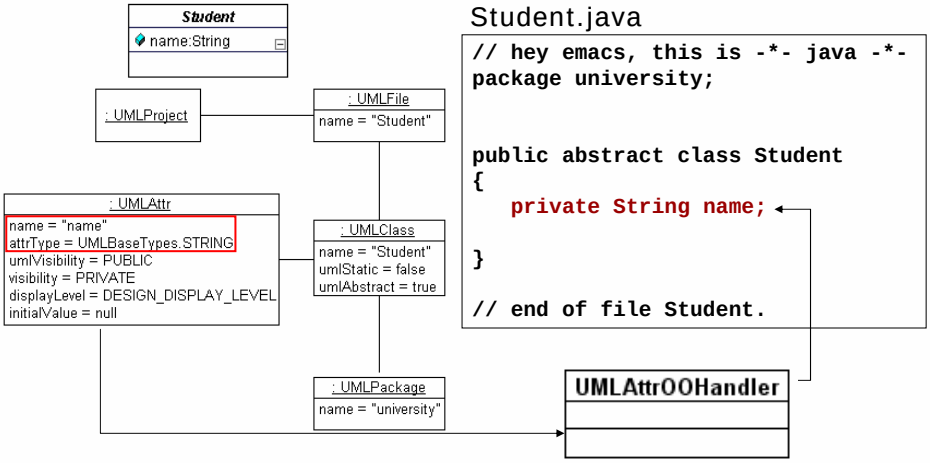
Generating classes – 3



Generating inheritance relationships



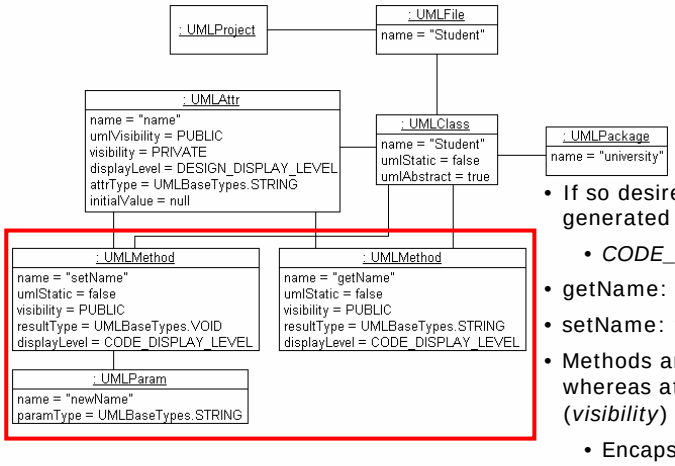
Generating attributes



Student.java

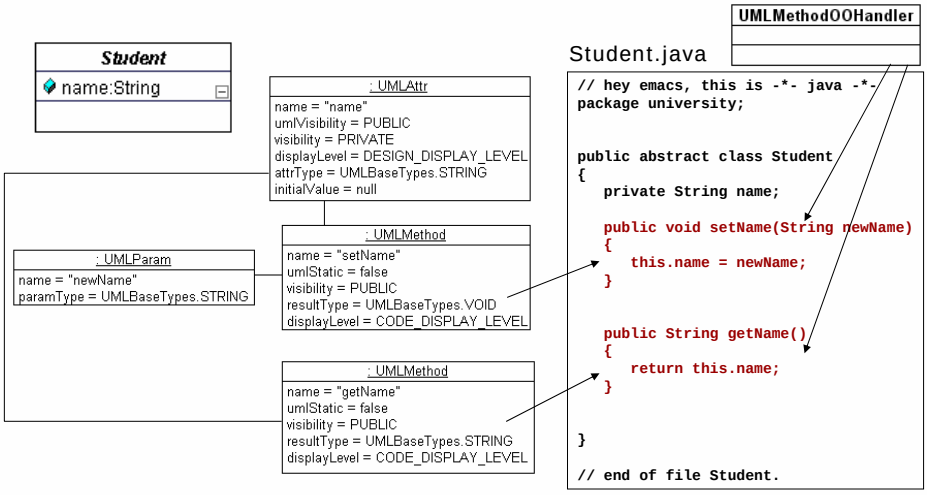
```
// hey emacs, this is -*- java -*-
package university;


public abstract class Student
{
    private String name;

}

// end of file Student.
```


Generating attributes – 2



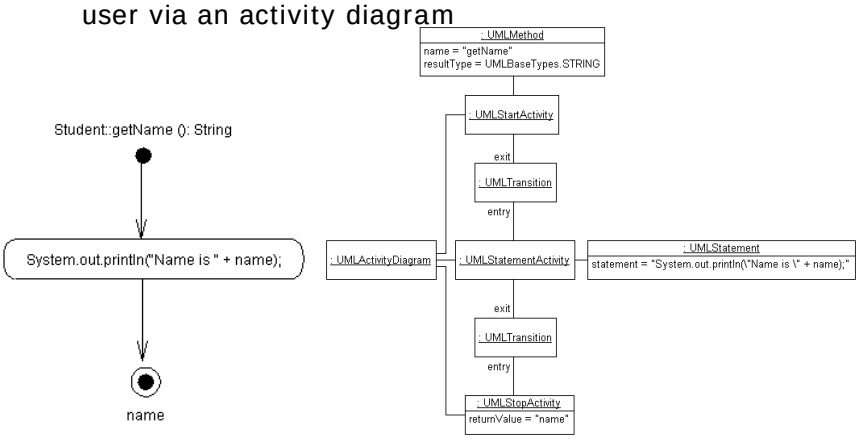
- If so desired, access methods are generated for every attribute
  - *CODE_DISPLAY_LEVEL*
- getName: `return this.name;`
- setName: `this.name = newName;`
- Methods are `public` (*umlVisibility*) whereas attribute is `private` (*visibility*)
  - Encapsulation

Generating methods



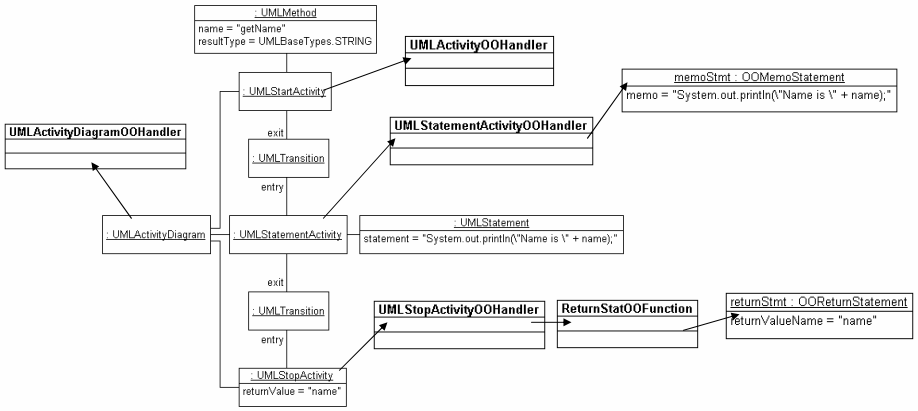
Generating method content from activity diagrams

- Suppose we had an alternative **`getName()`** specified by the user via an activity diagram

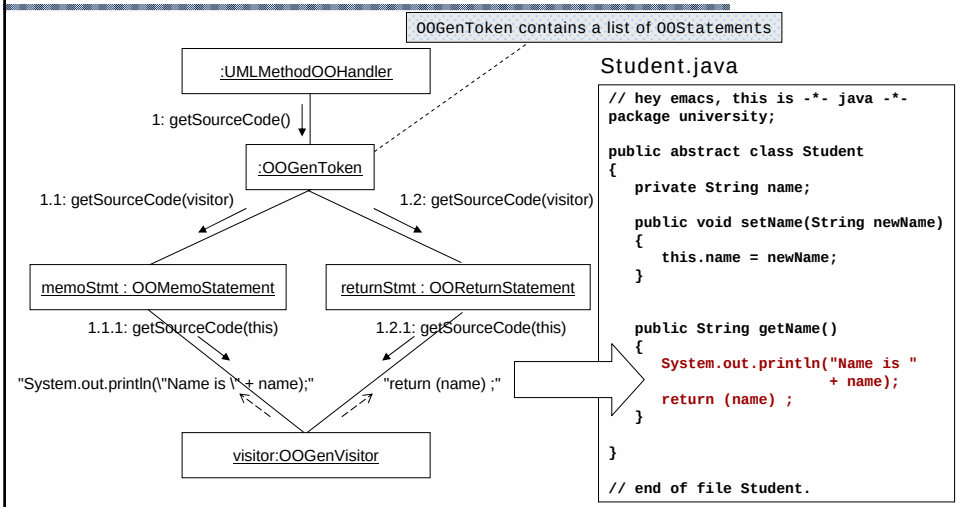


Generating method content from activity diagrams - 2

- Flow analysis performed on diagram to ensure correct order of code generation, then `UMLActivitys` are handled



Generating method content from activity diagrams - 3



Student.java

```
// hey emacs, this is -*- java -*-
package university;

public abstract class Student
{
   private String name;

   public void setName(String newName)
   {
      this.name = newName;
   }


   public String getName()
   {
      System.out.println("Name is "
                         + name);
      return (name) ;
   }

}

// end of file Student.
```

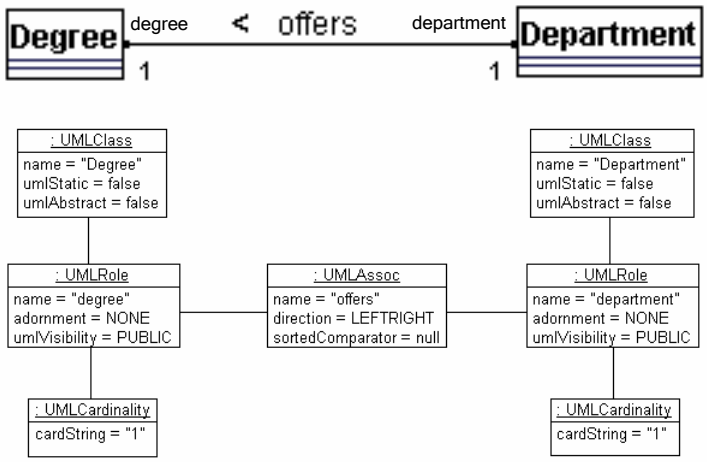
Overview

- ✓ Code generation mechanism design
- ✓ Generating classes, attributes and methods
- ➢ Generating associations
- ■ Making alterations to the mechanism
- ■ Summary and questions

Generating associations

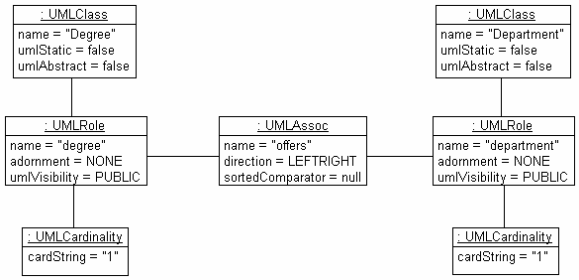
- Overview of the generation process
  - `UMLAttr` and `UMLMethod` objects (and the contents of the methods) are created from templates
    - `AssocCodeGenerator`
  - The normal mechanism for generating attributes and methods is then used
- Plan
  - Example of the generation process using a 1-to-1 association
  - Brief summary of differences in generating other types of association

1-to-1 association

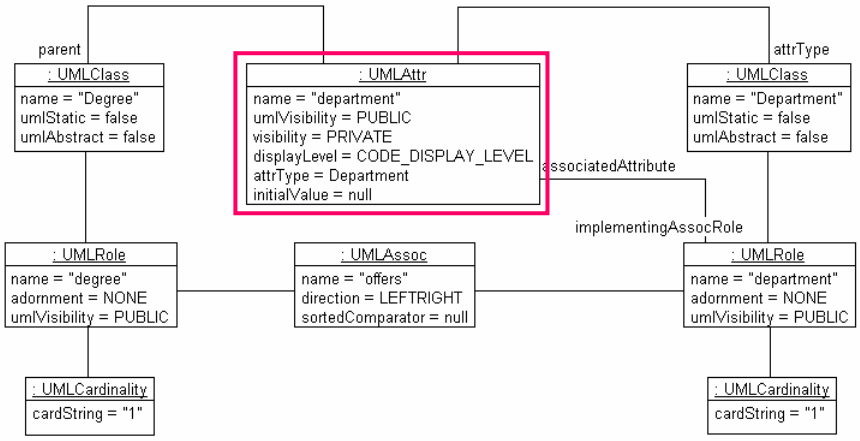


Analysing the roles

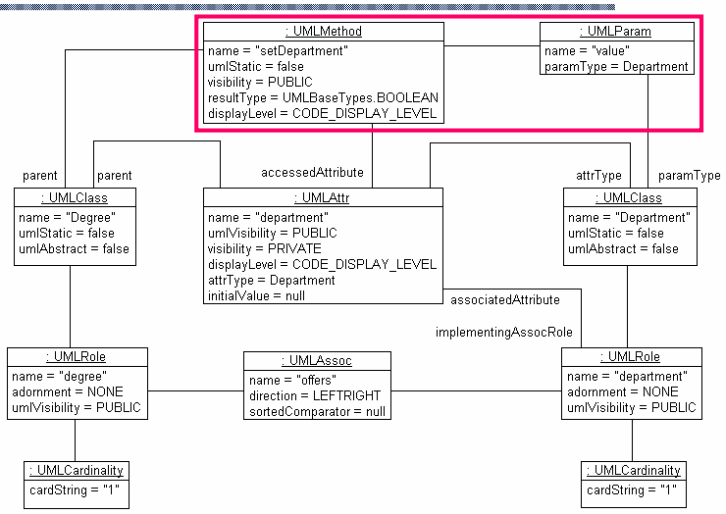
- Association is not qualified
  - no `UMLQualifier` objects
- `department`
  - Part of a bidirectional association
  - To-One role
    - `cardString == "1"`
    - so `Degree` is given
      - `private Department` attribute
      - `public` `set-method`
      - `public` `get-method`



Analysing the roles – 2



Analysing the roles – 3



Creating code for the methods

public boolean setRightRole ($RIGHTCLASS$ value)

```
#BeginCodeBlock = assoc-set-v1
boolean changed = false;

if (this.$RIGHTROLE$ != value)
{
   if (this.$RIGHTROLE$ != null)
   {
      $RIGHTCLASS$ oldValue =
    this.$RIGHTROLE$;
      this.$RIGHTROLE$ = null;
      oldValue.$REMOVE$;
   }
   this.$RIGHTROLE$ = value;
   if (value != null)
   {
      value.$INSERT$;
   }
   changed = true;
}
return changed;
#EndCodeBlock
```

- Association Template
- **$RIGHTROLE$** = department
- **$RIGHTCLASS$** = Department
- **$INSERT$** = setDegree (this)
- **$REMOVE$** = setDegree (null)

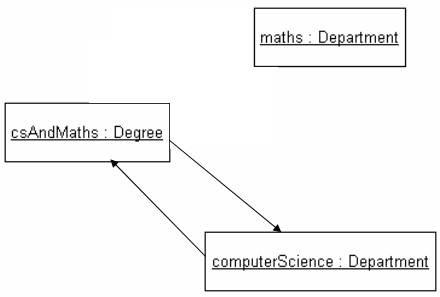
Creating code for the methods - 2

```
boolean changed = false;
if (this.department != value)
{
   if (this.department != null)
   {
      Department oldValue = this.department;
      this.department = null;
      oldValue.setDegree (null);
   }
   this.department = value;
   if (value != null)
   {
      value.setDegree (this);
   }
   changed = true;
}
return changed;
```

- $RIGHTROLE$ = **department**
- $RIGHTCLASS$ = **Department**
- $INSERT$ = **setDegree (this)**
- $REMOVE$ = **setDegree (null)**

Bi-directional associations

- Consistency must be maintained




```
public boolean setDepartment(Department value)
boolean changed = false;
if (this.department != value)
{
   if (this.department != null)
   {
      Department oldValue = this.department;
      this.department = null;
      oldValue.setDegree (null);
   }
   this.department = value;
   if (value != null)
   {
      value.setDegree (this);
   }
   changed = true;
}
return changed;
```


Differences in other association types

- Different data structures need to be used for the attribute, e.g.
  - To-Many associations - Hashset (`FHashSet`)
  - Qualified associations - Hashmap (`FHashMap`)
  - Ordered associations- Linked list (`FLinkedList`)
  - Sorted associations – Hashset (`FHashSet`) with comparator
- Different methods need to be added for access to the attribute
  - Reflect data structure and therefore association

Overview

- ✓ Code generation mechanism design
- ✓ Generating classes, attributes and methods
- ✓ Generating associations
- ➢ Making alterations to the mechanism
- ■ Summary and questions

Making alterations to the mechanism

- Possible scenarios
  - Handling new metamodel elements / handling metamodel elements differently
  - Different requirements for associations
    - e.g. Different container objects required, non-standard methods required
  - Generating a new language
- Combinations of the above are of course possible!

Changing metamodel element handling

- Create new handler(s)
  - Subclass of `OOGenStrategyHandler` or existing handler for element, e.g. `UMLProjectOOHandler`
  - Key methods are
    - **`OOGenToken generateSourceCode (ASGElement incr, OOGenToken prevToken, Object param[])`**
    - **`boolean isResponsible (ASGElement incr)`**
    - **`boolean needToken()`**
- Create new XML configuration file
  - Could obviously edit existing file but would thereby lose current configuration
- Parse configuration file
  - Call from plugin class which extends `AbstractPlugin`
  - **`CodeGenTargetParser.parse("mynewconfigfile.xml", getClassLoader());`**

Changing metamodel element handling – 2

- Enable new target
  - Example – new target name = "rtjava"

```
Vector v = JavaPreferences.get().getCodeGenTargetName();

String rtjava = "rtjava";
if (!v.contains (rtjava))
   v.add (rtjava);

String java = "java";
if (v.contains (java))
   v.remove (java);

JavaPreferences.get().setCodeGenTargetName (v);
```

Different requirements for associations

- Non-standard methods required?
  - Create new templates
- Create a new visitor
  - Key methods
    - To generate non-standard methods
      - **`public String getAssocTemplateName()`**
      - **`public String getReferenceTemplateName()`**
    - To use different container classes
      - **`public String getContainerName (OOCollectionEnum containerType, boolean bound)`**
  - If generating currently supported language, subclass current visitor and simply override above methods – otherwise subclass `OOGenVisitor`/`CodeGenVisitor`
- Create new XML file, parse and enable target

Generating a new language

- Create a new visitor
  - Subclass `OOGenVisitor`/`CodeGenVisitor` depending on requirements
  - Key methods
    - **`String getSourceCode([OOStatement classes])`** methods
      - `getSourceCode` methods visit the `OOStatement` classes - verify that appropriate code is generated from the abstract statement descriptions
    - Various class declaration, package declaration etc. methods
      - Specify or override where necessary
      - See `OOGenVisitor`/`JavaGenVisitor` `create*`, `generate*` methods
    - Template and container methods as before

Generating a new language - 2

- Create new XML file, parse and enable target
- **NB** Bear in mind that `OOStatements` only abstractly describe single statements. If the use of a new language requires a different sequence of statements, it will not be enough to simply replace the visitor – code using `OOStatements` must also be changed

Overview

- ✓ Code generation mechanism design
- ✓ Generating classes, attributes and methods
- ✓ Generating associations
- ✓ Making alterations to the mechanism
- ➢ Summary and questions

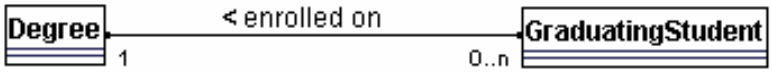
Summary

- Main classes in fundamental code generation mechanism and their relationships
- Generating code from static models and brief look at dynamic models
  - classes, attributes, methods and method content
- Building associations through addition of attribute and method objects
- Outline of starting points when changing the code generation mechanism in a variety of situations

- Written guide to workings of code generation mechanism on the Fujaba website (under Developer Documentation)

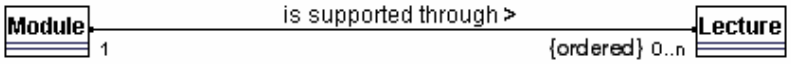
Extra Slides

To-Many associations



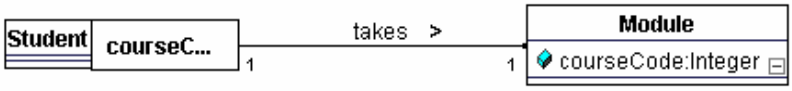
- Hashset used (`FHashSet`)
- Following methods are added to class `Degree`
  - **`hasInGraduatingStudent`**
  - **`iteratorOfGraduatingStudent`**
  - **`sizeOfGraduatingStudent`**
  - **`addToGraduatingStudent`**
  - **`removeFromGraduatingStudent`**
  - **`removeAllFromGraduatingStudent`**
- Class `GraduatingStudent` gets `setDegree` and `getDegree` methods
- Same for composition and aggregation associations

Ordered and Sorted



- Linked list used (`FLinkedList`)
- As well as the extra methods for To-Many associations, the following methods are added to the class `Module`
  - **`getLectureAt`**
  - **`indexOfLecture, lastIndexOfLecture`**
  - **`iteratorOfLecture(Lecture lowerBound)`**
    - Returns an iterator for all objects following `lowerBound`
  - **`isBeforeOfLecture, isAfterOfLecture`**
  - **`getFirstOf, getLastOf`**
  - **`getNextOf, getPreviousOf`**
  - **`getNextIndexOf, getPreviousIndexOf`**
- For a {`sorted`} association, a set (`FHashSet`) with a comparator is used, and only the methods **`getFirstOf`** and **`getLastOf`** are added

Qualified associations



- To-One associations use FHashMap
- All qualified associations have the following methods
  - **`hasKeyInModule, iteratorOfModule, keysOfModule, entriesOfModule, sizeOfModule`**
- The contents of the following methods, which we have seen before, is different
  - **`hasInModule, addToModule, removeFromModule, removeKeyFromModule, removeAllFromModule`**
- For To-One associations, the method **`getFromModule`** is also added

Story Driven Modeling and programming with Fujaba

Ira Diethelm
Gaußschule
Löwenwall 18a
38100 Braunschweig
ira.diethelm@uni-kassel.de

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

Albert Zündorf
FPM, Universität Kassel
Wilhelmshöher Allee 73
34121 Kassel
zuendorf@uni-kassel.de

ABSTRACT

The Unified Modeling Language UML has become the standard language for object-oriented modelling and documentation of software projects. The CASE tool Fujaba now uses the UML as visual programming language. Fujaba allows to do every phase of the software development process, from requirements analysis to testing, on the UML level. Fujaba not only defines a visual programming language but also provides tight tool integration for its own software development process called Fujaba Process (FUP).

In the FUP the developer systematically derives the complete software specification out of textual descriptions captured in the requirement phase. This is done in five different phases:

1. For each Usecase the developer writes textual descriptions of several scenarios describing this Usecase.
2. The textual descriptions have to be transformed into so called Storyboards. These are activity diagrams that contain object diagrams which describe the flow of the scenario comic strip like.
3. The classdiagram is derived (yet) manual but systematically out of the Storyboards.
4. For each Storyboard it is possible to generate a test method automatically which can be used to verify how many scenarios are already covered by the current implementation. The tests can also be used to ensure synchrones between scenarios and implementation.
5. By comparison and analysis of all Storyboards with a special eye on similar and alternative sequences we manage to create systematically (though manual) the behavioral description for each method that is involved in the scenarios. This procedure was first performed with students of a secondary school in Braunschweig, see [1, 2].
6. Fujaba can now generate standard Java code out which can then be compiled with a standard Java compiler.
7. The so generated program code can now be tested with our object browser and visual debugger Dobs.

Phase 2 to 5, the systematical derivation from Storyboards to a complete design specification is called Story Driven Modeling. In our tutorial we show how these systematically derivations are used and how easy method bodies can be programmed graphically with Fujaba.

Hope to see you.

1. REFERENCES

[1] I. Diethelm, L. Geiger, T. Maier, and A. Zündorf. Turning collaboration diagram strips into storycharts. In *Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE 2002*. Orlando, Florida, USA, 2002.

[2] I. Diethelm, L. Geiger, and A. Zündorf. Uml im unterricht: Systematische objektorientierte problemlösung mit hilfe von szenarien am beispiel der türme von hanoi. In *Erster Workshop der GI-Fachgruppe Didaktik der Informatik*. Bommerholz, Germany, 2002.