The FUJABA Environment

Ulrich Nickel
Computer Science Dep.
University of Paderborn
Warburger Str. 100
33098 Paderborn
Germany
+49 5251 603308
duke@upb.de

Jörg Niere
Computer Science Dep.
University of Paderborn
Warburger Str. 100
33098 Paderborn
Germany
+49 5251 603308
nierej@upb.de

Albert Zündorf
Computer Science Dep.
University of Paderborn
Warburger Str. 100
33098 Paderborn
Germany
+49 5251 603310
zuendorf@upb.de

1 OVERVIEW

The Fujaba environment aims to provide roundtrip-engineering support for UML and Java. The main distinction to other UML tools is its tight integration of UML class and behavior diagrams to a visual programming language. Our use of UML allows to model operations on graph-like object structures on a high-level of abstraction and leverages the user from programming with plain references at code level.

Code generation from class diagrams is widely known and supported by most modern case tools. However, code generation from class diagrams creates class frames and method declarations without bodies, where the actual work starts. Only a few case tools provide code generation for statecharts, e.g. [RR-RT, Rhap] and Fujaba. Statecharts specify the reactions of active objects on events in terms of state changes and in terms of actions annotated to states and transitions. These actions are usually pseudo code or program code, only.

Fujaba extends these capabilities by generating code from collaboration diagrams. Generating code from collaboration diagrams is easy if one restricts the employed collaboration messages to a single method body and if one uses only these messages for code generation. However, such collaboration diagrams require a lot of detailed collaboration messages. To overcome this problem, Fujaba leverages the user from providing a lot of detailed messages by assigning a standard semantics to the graphical elements of collaboration diagrams, cf. [KNNZ00]. This standard semantics covers a lot of navigational operations, that identify the collaboration participants, and a lot of operations for structural changes. Thus, our standard semantics simplifies the use of collaboration diagrams for programming purposes, significantly, and it turns collaboration diagrams in a powerfull and easy-to-read visual programming construct.

However, a single collaboration diagram is usually not expressive enough to model complex operations performing several modifications at different parts of the overall object structure. Such series of modifications need several collaboration diagrams to be modeled. In addition, there may be different situations where certain collaboration diagrams

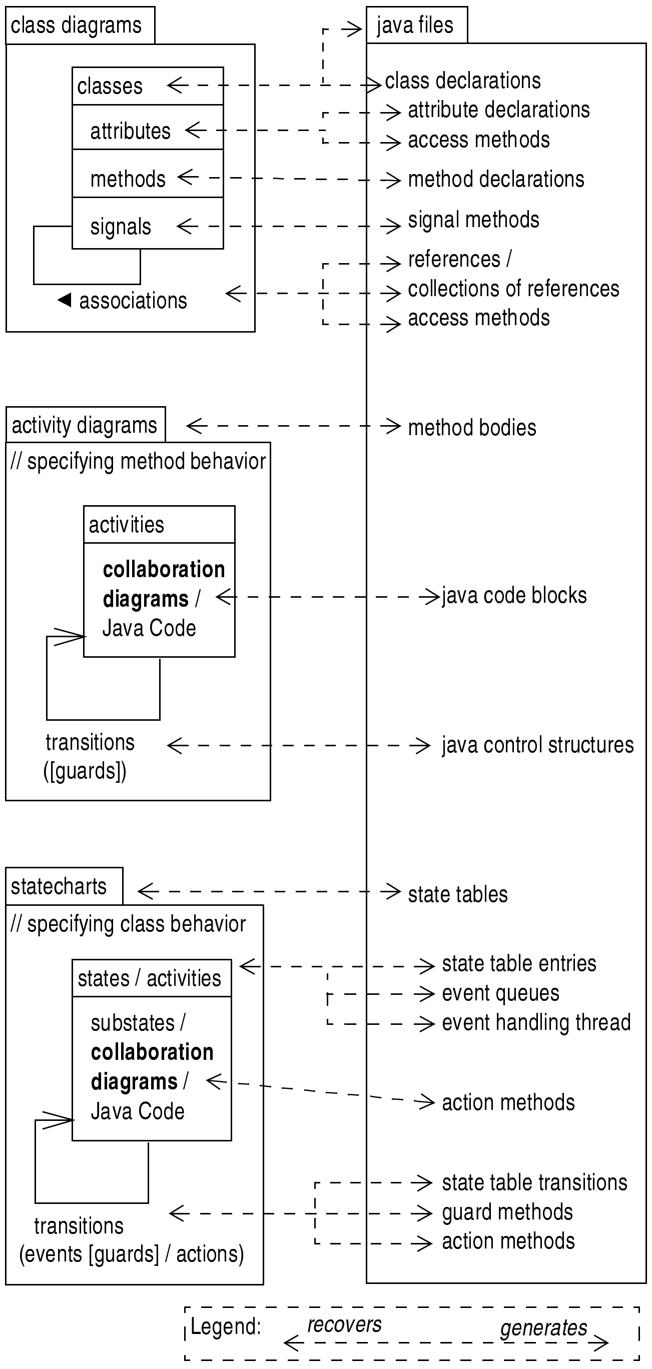


Figure 1 The Fujaba Environment Concepts

should be executed and others not. Thus, we need additional control structures to control the execution of collaboration diagrams. In our approach we combine collaboration diagrams with statecharts and activity diagrams for this purpose. This means, instead of just pseudo code, any state or activity may contain a collaboration diagram modeling the do-action of this step.

Figure 1 illustrates the main concepts of Fujaba. Fujaba uses a combination of statecharts and collaboration diagrams to model the behavior of active classes. A combination of activity diagrams and collaboration diagrams models the bodies of complex methods. This integration of class diagrams and UML behavior diagrams enables Fujaba to perform a lot of static analysis work facilitating the creation of a consistent overall specification. In addition, it turns these UML diagrams into a powerful visual programming language and allows to cover the generation of complete application code.

During testing and maintenance the code of an application may be changed on the fly, e.g. to fix small problems. Some application parts like the graphical user interface or complex mathematical computations may be developed with other tools. In cooperative (distributed) software development projects some developers may want to use Fujaba others may not. Code of different developers may be merged by a version management tool. There might already exist a large application and one wants to use Fujaba only for new parts. One may want to do a global-search-and-replace to change some text phrases. One may temporarily violate syntactic code structures while she or he restructures some code. For all these reasons, Fujaba aims to provide not just code generation but also the recovery of UML diagrams from Java code. One may analyse (parts of) the application code, recover the corresponding UML diagram(part)s, modify these diagram(part)s, and generate new code (into the remaining application code). So far, this works reasonable for class diagrams and to some extend for the combination of activity and collaboration diagrams. For statecharts this is under development.

The next chapters outline the (forward engineering) capabilities of Fujaba with the help of an example session.

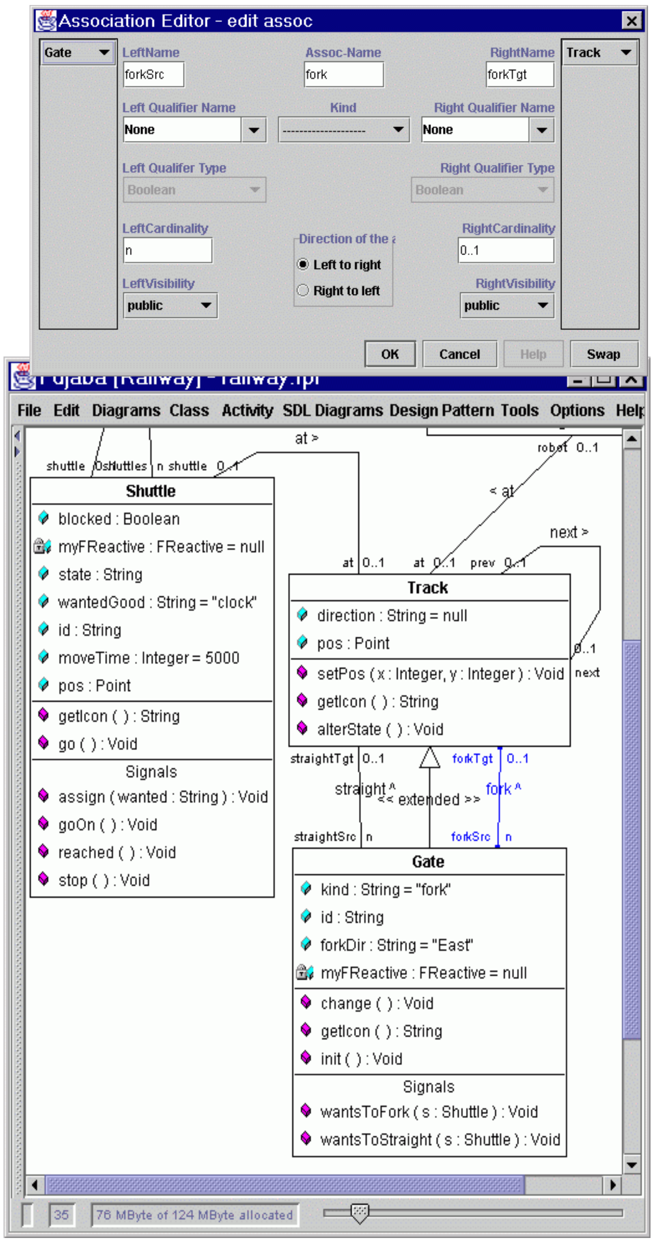
2 AN EXAMPLE SESSION

As running example for a short demonstration of the Fujaba environment we use the simulation of an automatic material transportation system in a manufacturing process. This example stems from the case-study of our ISILEIT project funded by the German National Science Foundation (DFG). This example employs autonomous transportation shuttles that travel on a track system.

Class Diagrams

Figure 2 shows the Fujaba environment editing a cut-out of the example class diagram. The shown association-dialog allows to create and modify all kinds of associations and to annotate the relevant detail information. Overall, the class diagram editor offers what one expects.

Figure 3 shows some details of the Java code generated for class Track and association fork. By default, associations are



Association Editor - edit assoc

Gate | LeftName: forkSrc | Assoc-Name: fork | RightName: forkTgt | Track

Left Qualifier Name: None | Kind: ------------------- | Right Qualifier Name: None

Left Qualifer Type: Boolean | Right Qualifier Type: Boolean

LeftCardinality: n | Direction of the a: (●) Left to right ( ) Right to left | RightCardinality: 0..1

LeftVisibility: public | RightVisibility: public

Figure 2 Fujaba showing the example class diagram

```
public class Track {
   private TreeSet forkSrc;
   public void addToForkSrc (Gate elem) {
      if ((elem != null) && !this.hasInForkSrc (elem)) {
         if (this.forkSrc == null) {
            this.forkSrc = new TreeSet (...);
         } // if
         this.forkSrc.add (elem);
         elem.setForkTgt (this);
      } // if
   }
   ...
} // class Track
```

Figure 3 Java code details for class Track and association fork

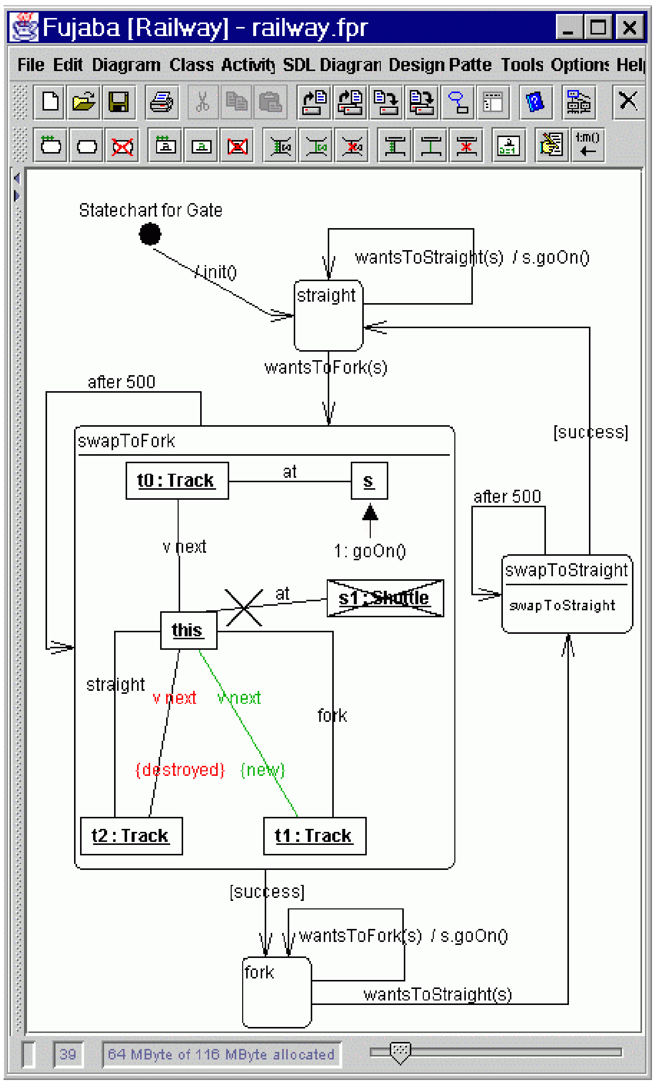


Figure 3 Details of the statechart of class Gate

implemented using pairs of pointers in both directions. To-many associations are implemented as collections of references, to-one associations employ usual references. We encapsulate the access to these (collections of) references, properly. Client programmers may establish links by just calling aTrack.addToForkSrc (aGate). The write access methods call each other, mutually (cf. Figure 3, line 9), this guarantees referential consistency. This reliefs the client programmer from taking care of the consistency of the pair of pointers that implement the desired association.

Statecharts combined with collaboration diagrams

Figure 3 shows some details of the statechart of class Gate. A gate is either in state straight or in state fork. When a shuttle arrives at a gate, it signals its desired direction via a wantsToStraight or wantsToFork event, respectively. Depending on its current state, the gate may have to swap its current direction first, cf. activity swapToFork. Finally, the gate sends a goOn event to the shuttle, signaling that it may proceed.

The body of activity swapToFork shows a collaboration

```
 1: public void doAction0SwapToFork () {
 2:    Object sdmTmpObject = null;
 3:    Track t0 = null; Track t1 = null; Track t2 = null;
 4:    boolean sdmSuccess = false;
 5:    try {
 6:       sdmSuccess = false;
 7:       // check link 'at' between this and s1
 8:       JavaSDM.ensure (this.getAt () == null);
 9:       t2 = this.getStraightTgt();          // bind t2 : Track
10:       JavaSDM.ensure (t2 != null);
11:       // check link 'next' between this and t2
12:       JavaSDM.ensure (this.getNext() == t2);
13:       t1 = this.getForkTgt ();             // bind t1 : Track
14:       JavaSDM.ensure (t1 != null);
15:       // check isomorphic binding
16:       JavaSDM.ensure (t2 != t1);
17:       t0 = this.getPrev();                 // bind t0 : Track
18:       JavaSDM.ensure (t0 != null);
19:       // check isomorphic binding
20:       JavaSDM.ensure (t2 != t0 && t1 != t0);
21:       // check link 'at' between s and t0
22:       JavaSDM.ensure (s.getAt() == t0);
23:       this.setNext (null);                 // delete link
24:       this.setNext (t1);                   // create link
25:       s.goOn();                            // 1:
26:       sdmSuccess = true;
27:    } catch (JavaSDMException sdmInternalException) {
28:       sdmSuccess = false;
29:    }  // try catch
30: }
```

Figure 4 Java code for the swapToFork activity

diagram modeling the swap operation. This collaboration diagram shows a cut-out of the object structure surrounding the current gate (the this object). The crossed-out shuttle object s1 models that the gate must not operate while it carries a shuttle. If this condition holds, the next link from this to track t2 is destroyed and a new next link from this to t1 is created, as indicated by the the {destroyed} and {new} constraints, respectively.

Figure 4 shows the Java code generated from activity swapToFork. Note, JavaSDM.ensure is a small library method, throwing an exception iff its argument is false. It is used to replace chains of nested if-statements with a single try-catch block. Note the use of association encapsulating methods for neighbor look-up and testing, e.g. in lines 8 and 9, and for modifications, cf. 23 and 24.

Graphical debugging and simulation

Fujaba focusses on modeling graph-like object structures. It does not yet include a graphical user-interface builder. (Integration with GUI builders is under development.) However, Fujaba provides a generic standard user interface, called Mr. Dobs, that shows a graphical view of graph-like objects structures and allows to call methods on objects, interactively. In Figure 5 one sees a small track system with three shuttles running. Note, shuttle s26 is currently blocked, since it wants to pass gate g20 straight but gate g20 had to wait until shuttle s24 had left it and is now about to swap its

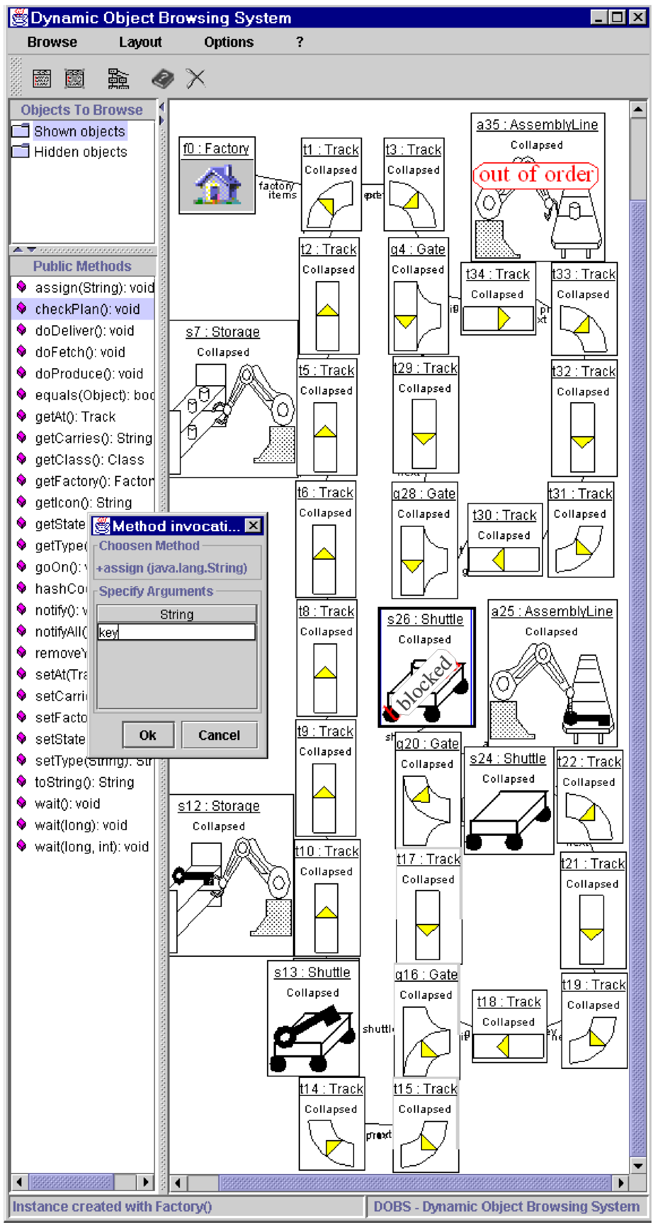


Figure 5 Simulation of a simple production system

direction.

The standard graphical user-interface Mr. Dobs works fine as an initial aid for testing a specification. It also allows to adapt the appearance of objects and to define more specific which objects should be visible and which should be hidden. In some cases, one may use this standard user interface as starting point for the development of the final user interface.

3 CONCLUSIONS

The development of Fujaba has started in December 1997. Since January 1999, code generation for class diagrams, activity diagrams, and collaboration diagrams and the generic standard user-interface are available. In 1999 we have added code generation for statecharts and round-trip engineering support for class diagrams. This work has already used Fujaba in a boot-strapping approach.

For the application to embedded systems, we have added an editor for SDL block and process diagrams and corresponding code generations. The SDL notation is very popular in the engineering areas. This work is close to its release. In addition, we currently extend Fujaba by an topology and deployment diagram editor, that will allow to model physical structures and the distribution of hardware devices and software agents operating these devices.

Urgent current work is round-trip engineering support for statecharts. Other current work tries to integrate graphical user-interface builders based on Java beans technology into Fujaba. This will enhance (or replace) the possibilities of our generic standard user-interface.

Fujaba already provides some round-trip engineering support for design patterns. This support will be extended by an PhD. thesis and several master thesis' during the next years. This will include the development of a new more sophisticated reverse engineering component for Fujaba. Another PhD aims to provide means for the integration of relational databases into modeled applications. Other long-term researches aims to extend Fujaba's support for the earlier development phases and for project managment.

The current prototype of the Fujaba environment is available as free software and comprises about 266 000 lines of pure Java code. The release of Fujaba is available via:

http://www.uni-paderborn.de/cs/fujaba/index.html

REFERENCES

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language User Guide*; Addison Wesley, ISBN 0-201-57168-4 (1999)

[FNTZ98]T. Fischer, J. Niere, L. Torunski, A. Zündorf: *Story Diagrams: A new Graph Rewrite Language based on the Unified Modelling Language and Java*; in Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, November 1998, LNCS, Springer Verlag, to appear (1999)

[KNNZ00]H. J. Köhler, U. Nickel, J. Niere, A. Zündorf: *Integrating UML Diagrams for Production Control Systems*; in Proc. Int. Conf. Software Engineering ICSE 2000, Limmerick, to appear, 2000

[Rhap] The Rhapsody case tool reference manual; Version 1.2.1, ILogix, http://www.ilogix.com/

[RR-RT] The Rational-Rose Realtime case-tool, http://www.rational.com

[SWZ95] A. Schürr, A. J. Winter, A. Zündorf. *Graph grammar engineering with PROGRES*. In W. Schäfer, Editor, Software Engineering - ESEC '95, LNCS 989, Springer Verlag, 1995.

[ZSW99] A. Zündorf, A. Schürr, and A. J. Winter: *Story Driven Modeling*, Technical Report, Universtiy of Paderborn, To appear 1999.