Testing and Simulating Production Control Systems Using the Fujaba Environment

Jörg Niere, Albert Zündorf

AG-Softwaretechnik, Fachbereich 17, Universität Paderborn
Warburger Str. 100, D-33098 Paderborn, Germany
e-mail:[nierej|zuendorf]uni-paderborn.de

Abstract. The Fujaba environment provides means for the specification of the software systems in UML notation and it has the opportunity to simultate the specified applications beforehand. Therefore, Fujaba provides editors for UML class diagrams for the static aspects of a software system and it provides Story Diagrams for the specification of dynamic behaviour. Story Diagrams combine UML activity diagrams for control flows and a UML collaboration diagram like notation for graph rewrite rules. Statecharts can be used for the specification of reactive objects. In Fujaba, each diagram has a precise formal semantics and this enables to generate Java code from the specification. The generated Java code is executed in a Java Virtual Machine (JVM) and can be visualized by an integrated object browser. This paper shows in a tool demonstration how to use the Fujaba environment in order to simulate a specification of a shuttle based production control system.

1 Introduction

This paper presents in a tool demonstration how the Fujaba[1] environment [FNT98] can be used to specify e.g. production control systems. Fujaba provides UML class diagrams for the specification of static parts of the system. A combination of UML activity and UML collaboration diagrams with an underlying graph grammar semantics for dynamic parts of software systems, e.g. methods. Statecharts [Har84] for reactive objects and SDL [ITU96] block diagrams for the specification of asynchronous messages. From such a specification, Fujaba is able to generate 100% pure Java code. The integrated Dynamic Object Browsing System (DOBS) allows a simulation of the generated code. Such a simulation enables us to test the specification in order to validate its behaviour.

2 Fujaba, an overview

Fujaba has been developed since 1998. Fujaba is planned as a round-trip and reverse engineering tools, that provides editors, code genrators and recognizers for all types of

1. From UML to Java And Back Again

UML diagrams. Figure 1 shows a screen shot of the Fujaba environment that contains the class diagram of the production control system example.

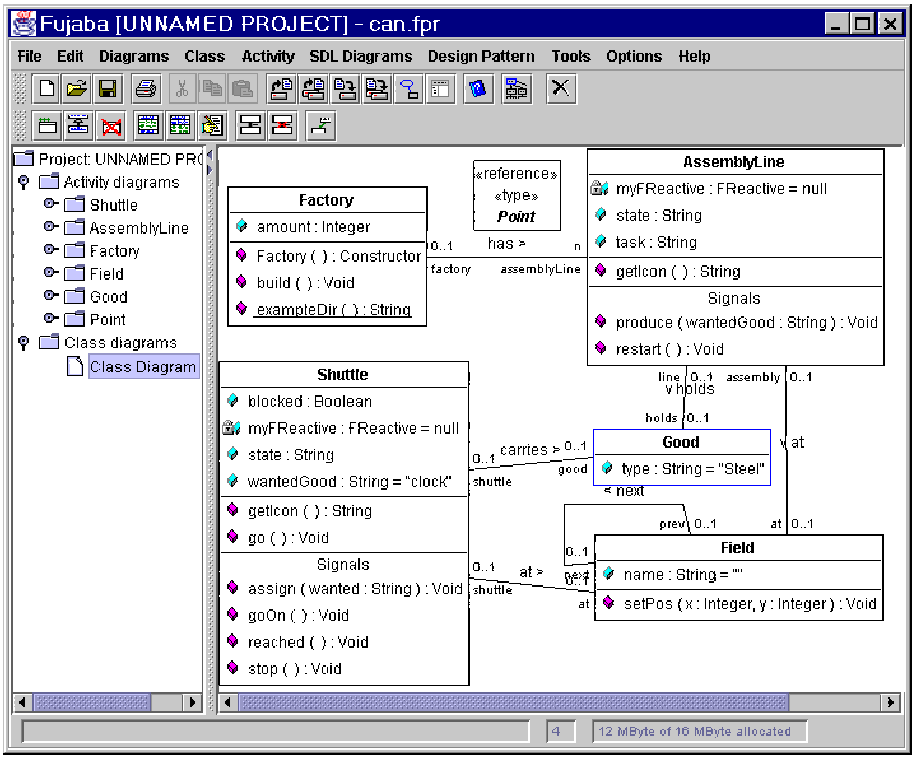

Figure 1 Fujaba class diagram screen shot

The class diagram shows a simple example specification of a production control system. Class Factory serves as a root class for the example and models the whole factory. The production line consists of fields represented by class Field. The self-association next connects fields to tracks for shuttles. Shuttles moving on fields are modeled by class Shuttle. Shuttles are able to stand at a certain field (association at to class Field) and to carry some good (association carries to class Good). The goods will be produced at assembly lines in the factory. Classes can contain attributes, e.g. the string typed variable wantedGood of class Shuttle, or methods and signals. In a class diagram, methods are only method declarations and method bodies can be specified using Story Diagrams. Fujaba combines UML activity diagrams and graph rewrite rules for the specification of method bodies. The control flow of methods is specified via UML activity diagrams, where each activity can contain either Java source code as well as graph rewrite rules.

Figure 2 shows a screen dump of Fujaba with the body of the go method of class Shuttle. The control flow of the method starts at the filled circle on the top of the diagram and follows the transitions. The first activity contains a graph rewrite rule. If the execution of the graph rewrite rule is successful, the control flow follows the transitions guarded with success and the method reaches the end-symbol. In case of a non-successful execution of the graph rewrite rule, the variable blocked is set to true and the method ends.

Graph rewrite rules in Fujaba show the left- and right-hand sides of a rule in a single picture and modifications are specified explicitly. We made the experience that typically rules look-up relatively large object (graph) structures, but contain only some modifications and so the single-picture notation is more compact and easier to read.

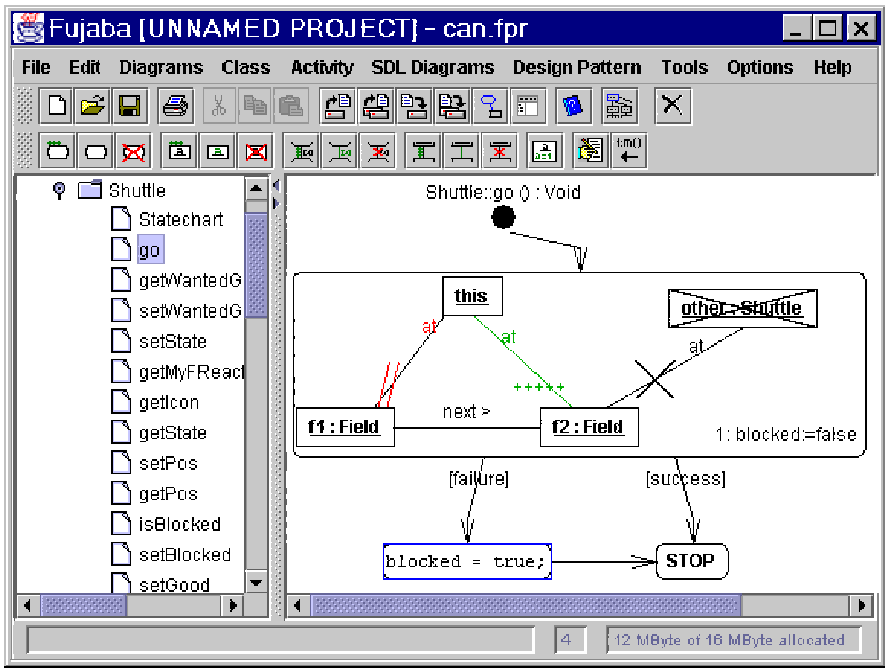

Figure 2 Story Diagram for method go of class shuttle

The execution of the graph rewrite rule works as follows: First bind each variable in the rule to an object in the object structure and then execute the modifications. Fujaba traverses the object structure starting from already bound objects and tries to bind the unbound objects of the rule. In Figure 2, the this object is already bound (the variable has no type) and thus, Fujaba first tries to bind the field object f1 by traversing the at link between this and f1. The next variable to be bound is f2, which is also a field. The cross-out of the at link between f2 and the variable other of type Shuttle specifies that there must not be another shuttle at field f2. Now, the modifications, specified in the rule, are executed, which means that the at link between this and f1 is deleted (specified by the two parallel, 'red' lines) and a new at link is created between this and f2 (specified by 'green' plusses). Finally, the collaboration statement "1: blocked := false" is executed.

3 The dynamic behaviour of shuttles

To specify the dynamic behaviour of reactive objects, Fujaba uses statecharts. Statecharts are assigned to classes, e.g. Figure 3 shows the statechart of class Shuttle. The notation is taken from the original statecharts introduced by Harel [Har84] and combined with graph rewrite rules.

The top-level states of a shuttle are waiting and active. After creation each shuttle is in state waiting and if an assign signal is received, the shuttle goes in the complex state active. The complex state active contains four different states, namely fetch, produce, sleeping and deliver. Each state represents a step in the production process of the sample factory. Like in Harel statecharts, each state consists of an entry-, a do-, and an exit-action. Guards, signals, and actions connected to transitions are notated as "signal [guard] / action". The execution sequence of actions is similar to Harel statecharts.

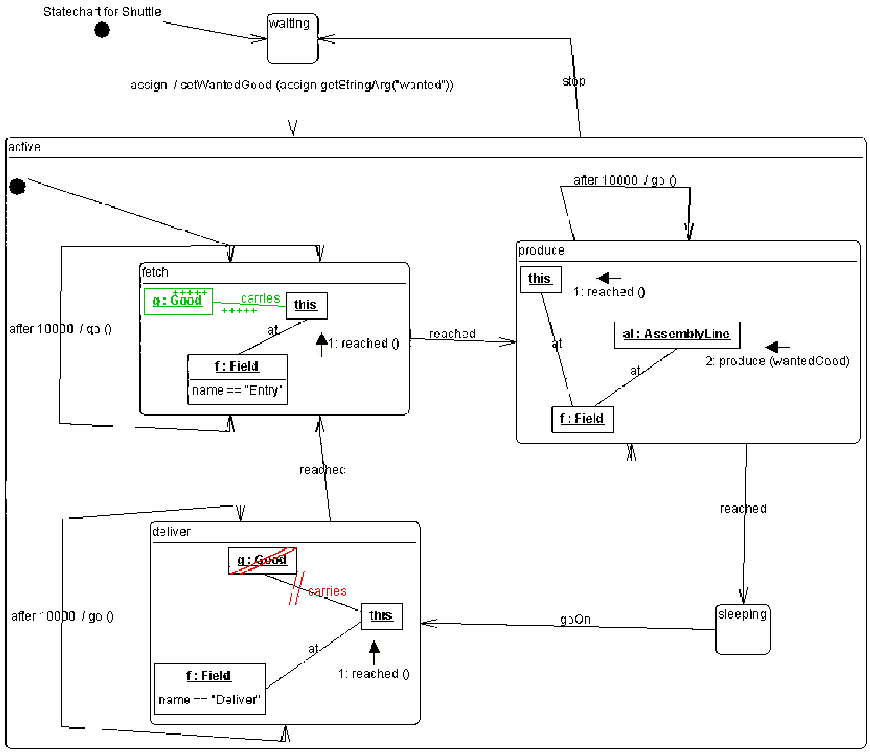

Figure 3 Statechart of shuttles

However, note that in our approach a statechart controls only a single reactive object, e.g. a single shuttle. Multiple reactive objects communicate with each other via asynchronous messages. This communication is modeled using SDL block diagrams.

State fetch of class Shuttle models that a shuttle has to go to a field named "Entry" and get some piece of iron. This behaviour is specified in the inline do-action, which is a graph rewrite rule. If the shuttle, specified by the this object, is currently not at a field named "Entry" the execution halts until a reached signal is received or a timer of ten seconds runs out. The latter is specified by the "after 10000 / go()" transition. The transition-action go lets the shuttle move to the next field (c.f. Figure 2) and the graph rewrite rule is executed again. Once the shuttle is at an "Entry" field a new Good object is created and put on the shuttle. After successful execution the shuttle sends itself a re-

ached signal, which is queued and after finishing the do-action the shuttle goes into state produce. The produce state checks if the shuttle is at a field with an assembly line. If not, the shuttle moves to the next field after ten seconds as in state fetch. When the shuttle reaches a field with an assembly line, it sends itself a reached signal and a produce signal to the assembly. After that, it goes into state sleeping and waits for a goOn signal. Once the shuttle receives a goOn signal from the assembly line, it switches to state deliver which models the delivery of the manufactured good. After delivery, the shuttle reaches state fetch again and the production process starts, anew.

4 Testing the specification

From the specified class diagrams, Story-Diagrams, and statecharts, the Fujaba code generator produces automatically 100% pure Java code. For example, classes are mapped to Java classes with attributes and method declarations. Story-Diagrams define the method bodies and the event handling of each class, which is specified by a statechart. Statecharts are implemented using threads in order to be executed concurrently. For more details of the code generation for class diagrams and Story-Diagrams see [FNT98, FNTZ99] and the mapping to threads is explained in [Koe99] in detail. The concurrency control concepts are taken from [Lea97].

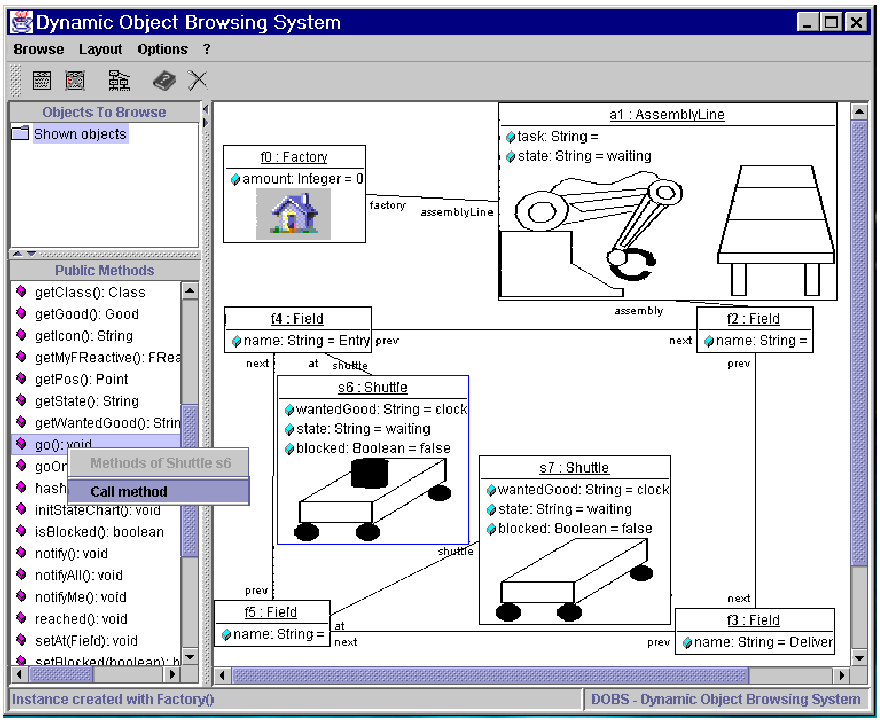

Figure 4 Current factory configuration

A sample factory configuration is shown in Figure 4. The screen shot is taken from the dynamic object browsing systems (DOBS) of Fujaba and shows the current object

structure within the Java Virtual Machine (JVM). There is a factory in the upper left and an assemby line in the upper right corner. The assembly line is standing at field f2 and the production line is currently a circle of four fields f2 to f5, which are connected via next/prev links. On the production line there are currently two shuttle s6 on field f4, and s7 on field f5.

Dobs allows to display attributes of objects based on the Java runtime type information and reflection mechanisms, for example the current attribute wantedGood has as value the string "clock". To provide a more convenient representation, Dobs can display different icons for objects. Therefore, Dobs uses the Java reflection mechanisms to ask an object, which icons shall be displayed. E.g. the icon of shuttle s7 shows only the shuttle itself, in contrast to shuttle s6, where a piece of iron is lying on the shuttle. This visualisation is more expressive than to show a good object with a carries link to the shuttle like it is modeled in the class diagram (see Figure 1).

The current factory configuration shall work as follows. Shuttles go to the field named "Entry" (field f4), pick up a piece of iron and then go to the assembly line. The assembly line produces a specific good out of the iron, e.g. a key or a lock and after that, the shuttle goes to a field named "Deliver", where the good is taken from the shuttle and stored (the storage is not modeled here).

To test this sample factory, Dobs allows the user to invoke methods on certain objects. For example the popup menu in Figure 4 shows that the go method of shuttle s6 is invoked by the user. So, for testing this specification, the user is e.g. able to let the shuttles do their job using the method invocation concept.

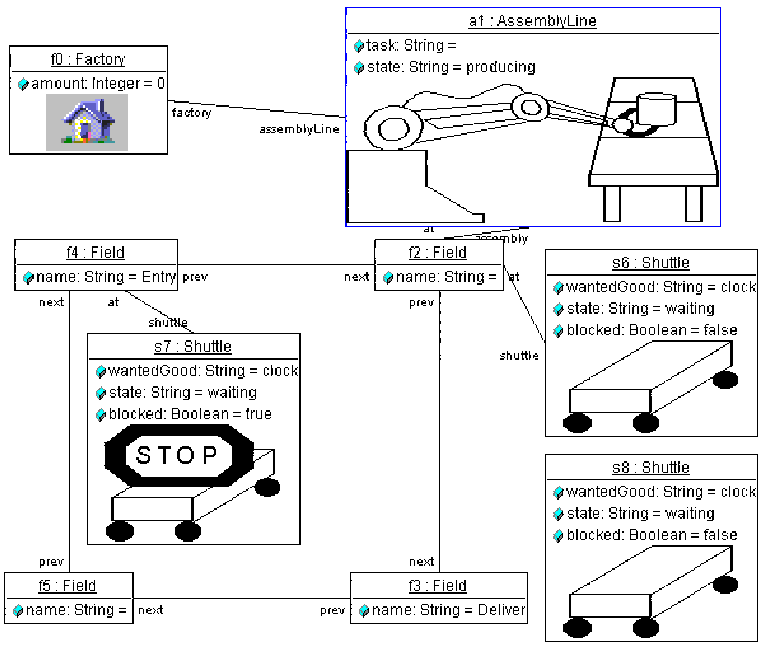

Figure 5 Assembly line producing a good

Such a test, where the user has to invoke different methods or doing layout, is very labor-intensive and the control parts rely on the user. But, reactive objects work autonomously and may be simulated in a continous way and not step by step. Therefore, the user just sends assign events to the shuttles. This turns the shuttle threads into state active and they start to execute the production process autonomously and concurrently.

Figure 5 shows the inlet of Dobs (see Figure 4) with the sample factory. The threads are just started and the shuttles are moving around. Shuttle s6 is currently at the assembly line, where a clock is produced out of the piece of iron the shuttles carrying in Figure 4. The stop sign in the icon of shuttle s7 says that the shuttle could not go on to the next field, because shuttle s6 is currently there, and so shuttle s7 is blocked[1] (see also method go of class Shuttle in Figure 2).

Now, there might be the need to reconfigure the production line, because the demands has changed or the production line is not as productive as it could be. Such reconfigurations can be done by the developer within the running simulation. For example, the production line can be extended by new fields and to raise the productivity an additional shuttle s8 might be installed. To facilitate reconfigurations, the system could be halted, but this is not necessary. Using a simulation, bugs in the specification can be pointed out, e.g. whether shuttles might crash or may block each other. But not only bugs in the specification can be pointed out, but also configuration problems. For example, if shuttles are mostly blocked, because the assembly line is to slow, the option of buying a second assembly line can be analysed by a simulation or other more optimal configurations can be discussed.

5 Conclusions and perspectives

We have shown, how the Fujaba environment can be used in order to specify production control systems and to test the specifications through simulation. The main problem is that modern production systems underly frequent changes and reconfiguration of hardware and software takes much time, because the software can’t be tested up front. We presented a possible solution to overcome these main problems by using the Fujaba environment. Fujaba has either the opportunity to edit a specification for a production control system as well as the generation of Java code and the simulation using Dobs.

Future work is to improve the simulation features of Dobs. For example, we need a script language for the configuration of Dobs, where parts of the configuration can be placed directly in the specification. Attributes shall be assigned with drawing objects, where e.g. a lamp attribute is displayed as a lamp and not as a text which says true or false. Another current project is a topology editor. E.g. a track-based production system may be plugged together in an editor offering track icons. This topology will be translated into graph rewrite rules to create initial starting object structures for the simulation afterwards.

---

1. attribute blocked is true

References

[FNT98] T. Fischer, J. Niere, L.Torunski, Conception and realisation of a integrated development environment for UML, Java and Story Driven Modeling (in german), Mater Thesises, Paderborn, 1998.

[FNTZ98] T. Fischer, J. Niere, L.Torunski, Story Diagrams: A new Graph Rewrite Language based in the Unified Modeling Language and Java, Proceedings of Theory and Application of Graph Transformations (TAGT), LNCS Spriner 1999

[Koe99] H.J. Köhler, Code generation for UML collaboration, sequence, and statechart diagrams (in german), Master Thesis University of Paderborn, to appear in 1999.

[Har84] D. Harel, Statecharts: A visual approach to complex systems, CS84-05, Department of Applied Mathematics, The Weizmann Institute of Science, 1994

[ITU96] ITU-T Recommodation Z.100 Specification and Description Language (SDL), International Union (ITU), Geneva, 1994 + Addendum 1996

[Lea97] D. Lea, Concurrent Programming in Java, Addison Wesley, 1998