Integrating UML Diagrams for Production Control Systems

Hans J. Köhler
Computer Science Dep.
University of Paderborn
Warburger Str. 100
33098 Paderborn
Germany
+49 5251 603310
hjk@upb.de

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

ABSTRACT
This paper proposes to use SDL block diagrams, UML class diagrams, and UML behavior diagrams like collaboration diagrams, activity diagrams, and statecharts as a visual programming language. We describe a modeling approach for flexible, autonomous production agents, which are used for the decentralization of production control systems. In order to generate a (Java) implementation of a production control system from its specification, we define a precise semantics for the diagrams and we define how different (kinds of) diagrams are combined to a complete executable specification.
Generally, generating code from UML behavior diagrams is not well understood. Frequently, the semantics of a UML behavior diagram depends on the topic and the aspect that is modeled and on the designer that created it. In addition, UML behavior diagrams usually model only example scenarios and do not describe all possible cases and possible exceptions.
We overcome these problems by restricting the UML notation to a subset of the language that has a precise semantics. In addition, we define which kind of diagram should be used for which purpose and how the different kinds of diagrams are integrated to a consistent overall view.

Keywords
UML, SDL, collaboration diagrams, statecharts, graph grammars, embedded systems

1 INTRODUCTION

Current production systems e.g. within factories for cars or any other complex industrial good face two major problems. First, production control software needs to become (more) dezentralized to increase their availability. It is not acceptable, that a failure of a single central production control computer or program causes hours of down-time for the whole production line. Second, todays market forces demand smaller lot sizes and a more flexible mixture of different products manufactured in parallel on one production line.

These problems may be solved by flexible, autonomous production agents. Some of these production agents might control specific parts of the overall production system like a single manufacturing cell or a transport robot. Other production agents may take the responsibility for manufacturing certain kinds of goods. Such flexible, autonomous production agents need knowledge of manufacturing plans for different goods and of their surrounding world, e.g. the layout of the factory or the availability of manufacturing cells. In addition, such production agents have to coordinate their access to assembly lines with other competing agents.

Figure 1 Snapshot of a production system

Common specification languages for embedded systems, like SDL [ITU96] and statecharts [HG96], focus on the specification of (re)active components of production control systems like control units, actors (e.g. motors, valves), and sensors (e.g. switches, lightborders, pressure, and temperature sensors), and on the interaction of such reactive components via events and signals. They provide no appropriate means for the specification of complex application specific object-structures as required by the described autonomous production agents.

Common object-oriented modeling languages, like UML, support the modeling of complex application specific object-structures. However, UML focuses on early phases of the

Published in the Proceedings of the 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, pp. 241-251, ACM Press, June 2000.

sofware life-cycle like object-oriented analysis and object-oriented design, cf. [BRJ99]. Thus, UML behavior diagrams, like collaboration and sequence diagrams, usually model typical scenarios describing the desired functionality, only.

Our work focuses on the design and implementation phase. We are looking for executable specifications and code generators. We use SDL block diagrams for the specification of the agents of a production control system and their communication channels (signal routes). We use statecharts for the specification of the reactive behavior of the production agents. Instead of pseudo-code, we use a well defined subset of collaboration diagrams as a visual programming language for the specification of do, entry, exit, and transition actions within statecharts. In our use, collaboration diagrams gain a precise operational semantics taken from graph grammar theory. This precise semantics allow the automatic translation of such collaboration diagrams to object-oriented programming languages like Java or C++. Thus, this paper integrates SDL block diagrams, statecharts and collaboration diagrams to form an executable specification language that allows to specify reactive behavior as well as complex application specific object-structures. We use the simulation of a production control system as a running example within this paper. However, we are confident that our modeling approach suits well for other application areas that deal with collaborating reactive objects and that employ complex application specific object-structures.

The next chapter discusses related work in more detail. Chapter 3 introduces our modeling approach and our integrated specification language. Chapter 4 describes the simulation of a simple production process as a running example for this paper. Chapter 5 discusses the translation of class-diagrams to Java code as a basis for the translation of behavior diagrams. Chapter 6 describes our concepts for statecharts. This enables us to discuss our use of collaboration diagrams and their combination with statecharts in chapter 7. Our debugging and testing facility is introduced in chapter 8. Chapter 9 summarizes our work and outlines current and future perspectives.

2 RELATED WORK

Current approaches for the specification of production agents use SDL [ITU96] or statecharts [HG96]. SDL is very popular in the electrical and mechanical engineering community. SDL block diagrams are used to specify processes and channels between such processes as well as messages passed via these channels. The behavior of embedded system processes is specified either using SDL process diagrams or using statecharts. Both notations basically model finite state automatas which react on signals by executing some actions, sending signals, and changing to new states. Both languages have a well defined formal semantics and tool support for analysis and simulation and code generation is available [GK97, Doug98, H+88, SGW94, AT98, Rhap, RR-RT].

Compared to SDL process diagrams, statecharts provide more expressive language features like nested states, and-states, and history states. In addition, the modeling of the internal process behavior becomes the domain of software developers (instead of mechanical or electrical engineers) which are more used to statecharts than to SDL process diagrams. Thus, we decided to adopt statecharts for this purpose.

However, statecharts (as well as SDL process diagrams) lack appropriate means for the specification of the actual actions triggered by the received signals. Usually, one has to use pseudo code for this purpose and in case of code generation one actually deals with the nasty details of current textual programming languages. Statecharts provide sophisticated means for the specification of (concurrent) control flows for reactive objects. However, by purpose statecharts abstract from the complex application specific object structures that build up the concrete states of a system. Statecharts do not explicitly deal with values of attributes or with links to other objects nor with the evolution and changes of these object-structures caused by the execution of usual methods.

The specification of application specific object-structures is a well known application area for graph grammars, cf. [Roz97, SWZ95]. Basically, a graph rewrite rule allows the specification of changes to complex object-structures by a pair of before and after snapshots. The before snapshot specifies which part of the object-structure should be changed and the after snapshot specifies how it should look like afterwards, without caring how these changes are achieved. While graph grammars are appropriate for the specification of object-structure modifications, they lack appropriate means for the specification of control flows. Even the well known graph rewrite system Progres [SWZ95] provides only textual control structures.

To overcome this problem, in previous work, we introduced UML activity diagrams as high-level control flow notation for graph rewrite rules, cf. [JZ98, FNTZ98]. In order to facilitate the use of graph rewrite rules for object-oriented designers and programmers, we additionally adapted UML collaboration diagrams as a notation for object-structure rewrite rules. For this combination of activity diagrams and collaboration diagrams we use the name *story-diagrams*, cf. [JZ98, FNTZ98]. Story-diagrams are a powerful visual progamming language that provides ideal means for the specification of complex application specific object-structures. Story-diagrams are well suited for providing our production agents with an appropriate model of their surrounding world and for modeling complex manufacturing plans, cf. [NZ99]. However, story-diagrams still lack means for the specification of the reactive behavior of communicating and collaborating production agents. Thus, this paper proposes to extend story-diagrams by statecharts to so-called *story-charts*. Story-charts use statecharts (and activity diagrams) to specify complex control flows and collaboration diagrams to specify the entry, exit, do, and transition actions that deal with complex object-structures.

3 MODELING APPROACH

Once the general requirements engineering and analysis work is done, we propose the following steps for the specification of flexible production control systems, cf. Figure 2:

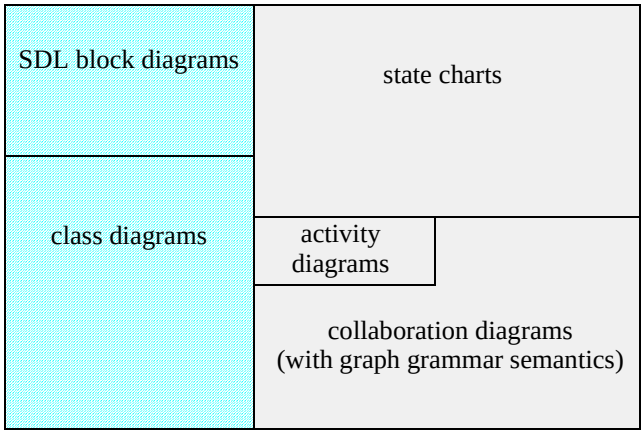

Figure 2 Diagram combination for the modeling approach

1. **Analysis Consolidation Phase**: Based on the results from the general analysis phases, in this phase the topology of the planned production control system is modeled. This includes the identification of the number and types of participating processes as well as the definition of communication channels and of all kinds of interchanged signals.
Within our projects we use SDL block diagrams for this purpose, since SDL block diagrams are very popular in engineering disciplines and this work is done in close collaboration with mechanical and electrical engineers. However, in other application areas one might use UML deployment diagrams or ROOM [SGW94] capsules and ports for the same purpose.

2. **Design Phase**: In the next step, one derives the initial (UML) class diagram of the desired system. At this, each process(type) identified in the SDL block diagram generates a class in the class diagram. In addition, each signal received by a process in an SDL block diagram creates a signal method in the corresponding class.

3. **Reactive Behavior Modeling**: SDL block diagrams (and the derived class diagrams) specifiy the number of signals which are understood by the different processes. Now, we have to define how each process will react on these signals. Thus, for each process class, one has to provide a statechart modeling the general process behavior. These statecharts should at least cover all signals that are understood by / declared in the corresponding process class.
In the engineering field, usually SDL process diagrams are used for this purpose. However, we prefer statecharts here, due to the additional expressive power of nested states and and-states.

4. **Actions Modeling**: First, statecharts specify at which states a certain process reacts on which signal. In response to a signal a process might change its state and execute some additional activities. For a flexible production agent, these activities might again include complex computations. These complex computations might employ or modify complex object-oriented data structures in order to reflect the surrounding world or the execution of manufacturing plans for certain products. For the specification of such complex computations we propose to use UML collaboration diagrams. Thus, we propose collaboration diagrams to specify complex control flows of methods employed as actions within statecharts. For operations on complex object- structures, we propose to use collaboration diagrams as a specification means.
Frequently, one will need more than a single collaboration diagram to model a number of object structure modifications. Therefore, we combine statecharts (and activity diagrams) with collaboration diagrams to a powerful visual specification language. Basically, we allow to use collaboration diagrams as the specification of activities instead of just pseudo code statements.

5. **Code Generation**: Once all aspects of the system are specified, the Fujaba environment is able to generate a complete, executable Java implementation from the class and behavior diagrams. Well, to be honest, process distribution and communication aspects defined in the SDL block diagrams are not yet covered by our code generator and need manual coding. This is current work. However, the currently generated code suffices for the following simulation phase.

6. **Simulation**: One of the main problems in todays manufacturing industry are long down times of assembly lines due to long testing phases on the installation of new software. Thus, we propose to simulate the production process beforehand in order to shorten software reconfiguration downtimes of physical assembly lines.

Note, during the specification of the process behaviors one usually will extend the class diagrams e.g. by attributes and associations and auxiliary classes modeling the object structures employed by the production control system. In addition, one will add auxiliary methods that encapsulate complex computations employed by statecharts in response to received signals. Actually, any item used in a behavior diagram has to be declared within the class diagram(s).

In our approach, the different diagrams are logically and mutually dependend on each other. The SDL block diagrams specify the minimal number of (process) classes to be contained in the class diagram and for each such class all its signal methods. In addition, each process class is equiped with one main statechart. This statechart has to define the response to all signals of the corresponding class. In addition to state changes, this response might include actions. Each of these actions is specified using one behavior diagram (which in turn may apply additional diagrams for subactivities). Thus, within the resulting overall specification each aspect of the system is specified by exactly one diagram and it is absolutely clear how these different diagrams are related to each other and how they form the whole specification. In addition, a specification is complete only if all aspects identified in the SDL block diagram are covered by class diagrams and statecharts and if in turn all auxilliary aspects introduced in these diagrams are covered by additional behavior diagrams (until transitive closure is reached).

Note, the described nesting of specification diagrams is enforced for the final system specification, only. One might use additional UML behavior diagrams in order to analyze certain scenarios. Such diagrams need not to be complete and there may be multiple diagrams that overlap in several aspects. One may study such scenarios at any time during the specification process and one should keep such diagrams for

documentation purpose. Actually, this is very important modeling work, too. However, in our approach, such additional diagrams are not part of the final, overall, executable system specification and they do not contribute to the code generation.

The next chapters discuss the phases of our approach in more detail.

4 RUNNING EXAMPLE

This paper uses the simulation of a simple production process as running example. This production process models a factory with various manufacturing places and with shuttles transporting goods from one manufacturing place to another. The example stems from the ISILEIT project funded by the German National Science Foundation (DFG). The goal of the project is the development of a formal and analyzable specification language for manufacturing processes. This specification language shall allow to verify important system properties like lifeness and the absence of deadlocks, e.g. via model checking. In addition, a code generator shall provide automatic code generation for the building blocks of a manufacturing process, namely shuttles, gates, storages, assembly lines, etc.

After the requirements engineering and analysis work, our approach proposes an *analysis consolidation phase*, where the topology and the block diagrams have to be specified.

Figure 3 shows the topology of the sample factory used as a running example. The production line consists of a track-based transport system, where shuttles are moving. In our example there are one main track circle and transfer gates to reach two assembly lines. We call either the assembly lines on the right as well as the storages on the left stations. Stations and their related robots build an operating unit, e.g.

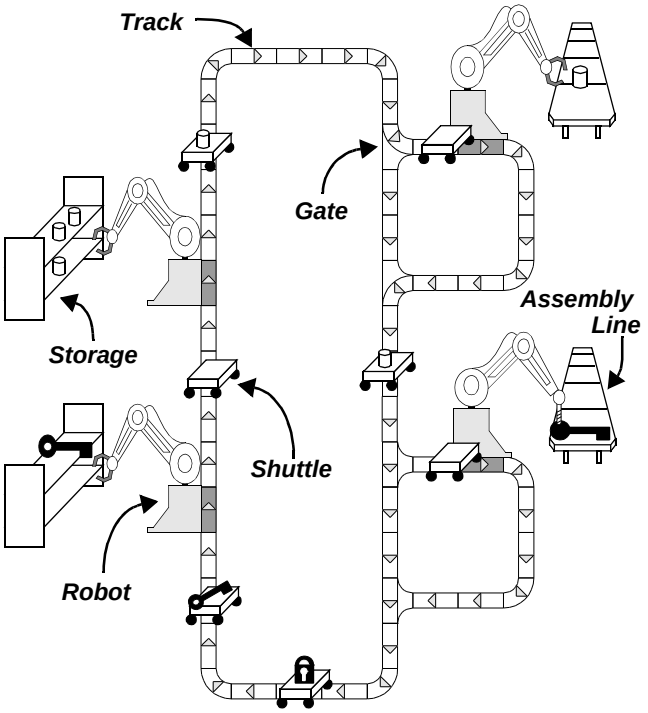

Figure 3 Topology of our sample factory example

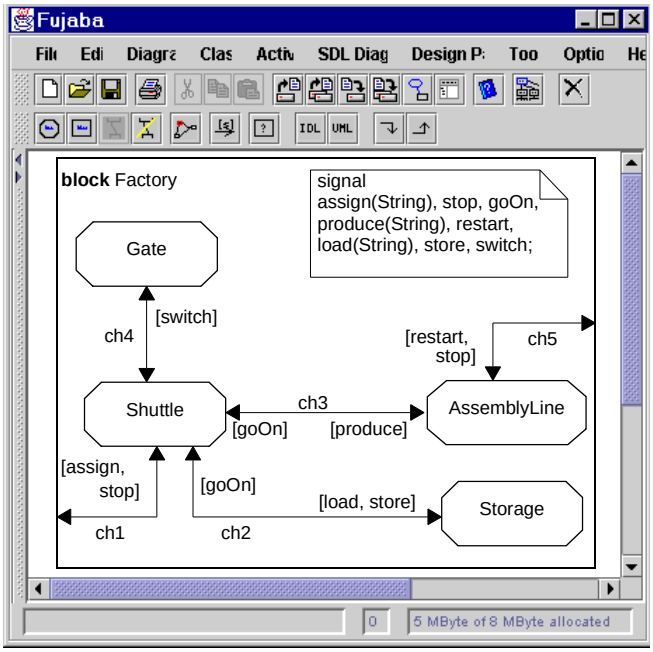

Figure 4 SDL block diagram of the sample factory

at storages steel can be loaded on shuttles or manufactured goods can be taken from shuttles. Assembly lines are able to manufacture different kinds of goods, e.g. keys or locks, out of a peace of iron.

Seven shuttles are currently working on the track system. Each shuttle executes a defined working task. This task starts, when a shuttle is assigned to produce a certain good, e.g. locks. The first step in the working task is to collect a piece of iron from the upper left storage, then it has to move to an assembly line and to order the wanted good. At each transfer gate the shuttle has to decide where it wants to go according to the choice of the assembly line and the shortest path to get there[1]. Once the shuttle has reached an assembly line, it orders its assigned good. The related robot takes the piece of iron from the shuttle (e.g. upper right assembly line), and manufactures the good, using different tools (e.g. lower right assembly line). The produced good is put on the shuttle and has to be brought to a storage (lower left storage). After the shuttle has moved to the storage and the good is stored, the working task has reached the end and the shuttle starts again from the beginning. The shuttle will execute this tasks until it gets a new assignment or it "dies".

From this example topology and overall behavior one may derive an SDL block diagram by identifying participating processes and communications between them.

For our sample factory, the processes are shuttles, gates, assembly lines, and storages. These processes can be derived from the description above, where shuttles are the central parts communicating with other parts in the factory in order to fulfill their working tasks. Figure 4 shows a screen-shot of the Fujaba environment [FNTZ98] with the SDL block

1. The decision depends on the current tool, and on the number of waiting shuttles at the assembly line.

diagram for the factory example. It contains the four processes *Gate*, *Shuttle*, *Storage*, *AssemblyLine* and communication channels (*ch1...ch5*). Channels connecting two processes model a communication between the processes, and channels going from a process to the border of the diagram model a communication with an outer block, here the user. Communication signals are specified in square brackets at the end of an arrow head. For example, the *AssemblyLine* process can either receive a *produce* signal from the *Shuttle* process as well as a *restart* or *stop* signal from the user. The *Storage* process models both types of storages, one to get a piece of iron and one to store the manufactured good. Therefore, a *Storage* can receive a *load* and a *store* signal. As the signals at the channels are only named, their declaration including parameters must be specified in a so-called comment frame titled *signal* as shown in the upper right corner. For example, an *assign* signal is declared with a string parameter for the good a shuttle shall 'produce'.

Note, in this work we use SDL block diagrams as a kind of use-case diagrams that model the services provided by the different processes. Alternatively, one could use UML deployment diagrams for the same purpose.

From such SDL block diagrams the Fujaba environment [FNTZ98] derives an initial class diagram with process classes for each identified process (kind) and signal methods for each signal understood by these process classes [San99]. In addition, we would like to generate code that initializes the production control system at boot time and creates the employed production agents and establishes the communication channels. This is current work.

5 FROM CLASS DIAGRAMS TO JAVA CODE

The next phase in our modeling approach is the *design phase*, where each process of the SDL block diagram generates a class in the initial class diagram. The signals in the block diagram become signal-methods of the corresponding classes. During further development this class diagram is extended to model additional structures like attributes, associations and auxiliary classes. Figure 5 shows a screen-shot of the Fujaba environment with the UML class diagram for the production process example. It contains classes like Shuttle, Storage, AssemblyLine, and Gate that are derived from the SDL block diagram. The auxiliary class Track is used to model the topology of our factory. The classes Storage and AssemblyLine inherit from class Station, which inherits from class Track. This inheritance associates these components with certain tracks. In addition, class Gate is specialized into Splits and Joins, which implement different swap methods for switching between outgoing or incoming tracks, respectively.

The translation of class diagrams to an object-oriented programming language is straight-forward and provided by most current OO-CASE tools, [RR-RT, Rhap]. The Fujaba generator [FNTZ98] translates UML classes to Java classes. Methods are translated into Java method declarations. The method bodies are usually empty (but a designer may specify a body using UML behavior diagrams see chapter 6 and 7). According to the Java Beans style guides, we translate

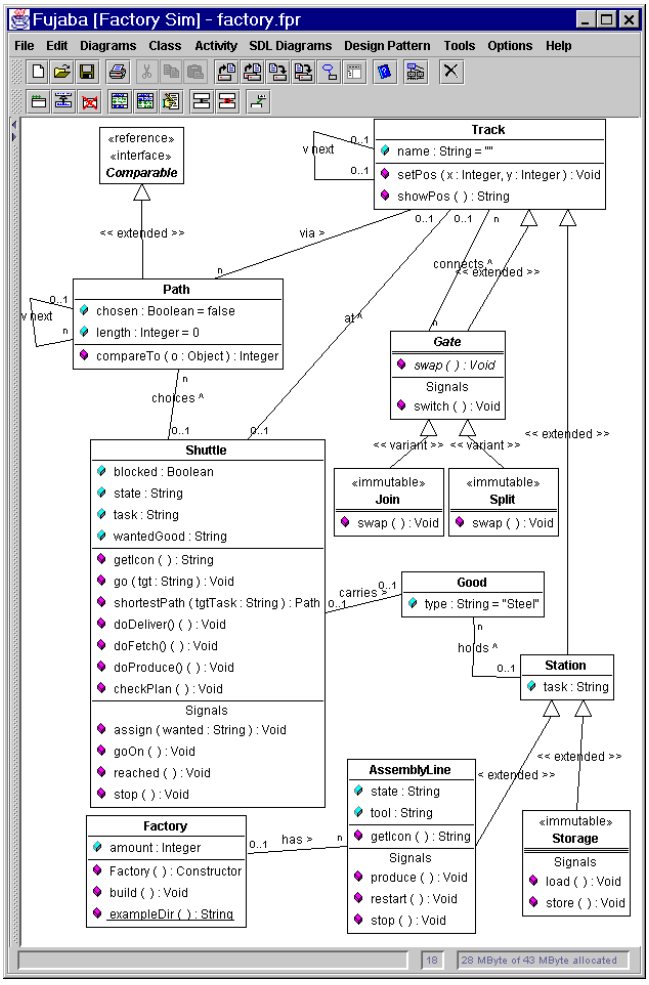

Figure 5 Fujaba showing the example class diagram

attributes to private Java attributes accessible via appropriate get- and set-methods.

UML associations are usually bi-directional. Thus, we implement associations by pairs of references in the respective classes. For multi-valued associations, like the (reverse direction of the) choices association between Path and Shuttle, we use standard container classes provided by the Java Foundation Classes [JFC99]. In order to guaranty the consistency of the pairs of references that implement an association, the respective access methods for reference attributes call each other. Overall, the implementation of associations is close to the strategy of the Rhapsody tool [Rhap].

The generation of code for UML class diagrams provides the basis for the translation of behavior diagrams within the next phases of our modeling approach.

6 REACTIVE BEHAVIOR VIA STATE-CHARTS

For the *reactive behavior modeling* we use statecharts. Statecharts can be used for many different purposes. They can model the behavior of a whole application or just a single method. Statecharts may model all allowed sequences of method invocations on a certain object or they may model state dependent reactions of certain objects on the reception

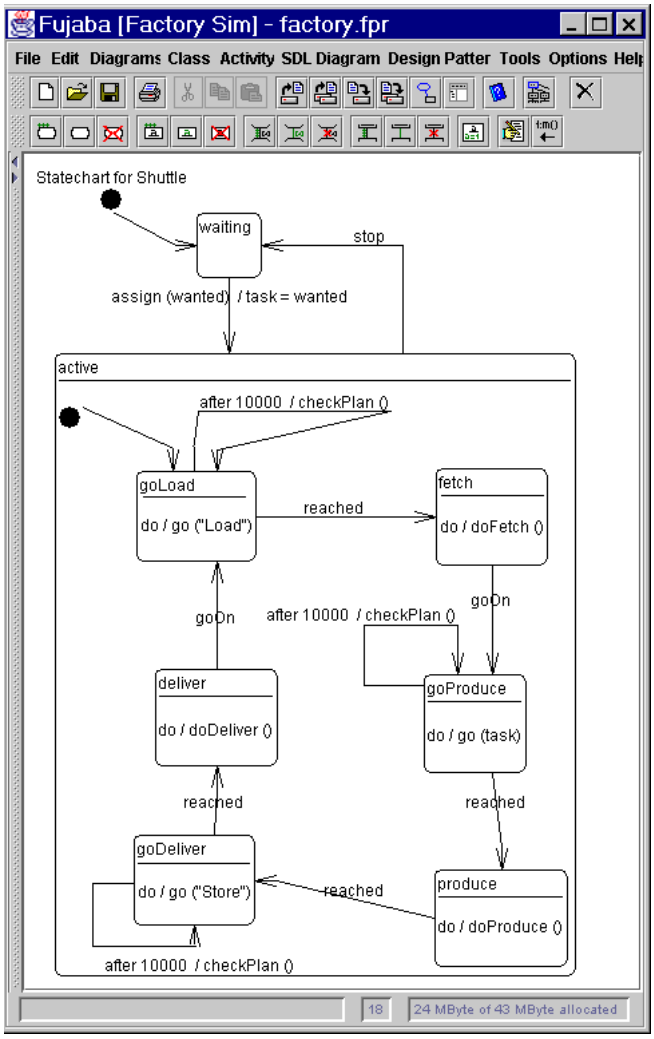

Figure 6 Simple State-chart of a shuttle

of certain events. Each of these uses leads to different semantics of statecharts. In addition, the different usages employ different statechart language features. In order to generate code from a state-chart one must clearly identify which part of an application and which behavioral aspect is modeled by the provided statechart.

This chapter discusses the use of statecharts for the specification of the reactions of production agents to signals invoked by other agents in the production system. Thus, in our approach statecharts specify the reactive behavior of process classes that are derived from the SDL block diagrams. Thus, depending on the object's current state a statechart defines which actions will be executed and which successor state is entered.

Figure 6 shows a very simple statechart for class Shuttle that is used to outline the general behavior of a shuttle and that allows some simple simulations. In this simulation, class Shuttle models the key players in our production control system. Shuttles fetch material from certain storages, travel to appropriate assembly lines, initiate manufacturing steps, and deliver the produced goods at other storages.

Shuttles have two top level states. Initially, they are in state waiting where they accept assign events. As declared in the SDL block diagram and in the UML class diagram, an assign event has one parameter wanted which describes the good to be produced by a shuttle. Thus, the shuttle stores this value in its task attribute and switches into state active. While a shuttle is active, at any time a stop event will stop it and it changes back into state waiting. State active is a complex state that controls the manufacturing process executed by a shuttle. Basically, a shuttle loops through the three main production phases fetch material, produce the desired good with the help of an assembly line, and deliver the manufactured good at the storage.

Each phase consists of two steps. First the shuttle travels to an appropriate station. For example, the do method of state goLoad calls activity go(„Load"). The go activity itself is a complex operation specified by a sub-statechart. This sub-statechart employs methods to compute the shortest path to the target station and it deals with gates that need to be switched and it operates the physical sensors and actors of the shuttle. Its details are omitted due to the lack of space. Once the go activity reaches the target station the shuttle sends itself a reached event that issues the next production step, cf. Figure 6. While the shuttle is traveling, every 10000 milliseconds it reconsiders the current situation via method checkPlan. Method checkPlan checks the state of gates and shuttles and stations it is passing or traveling to. In case of problems it may change plans. If for example the target station is out-of-order it might switch to an alternative station.

In order to simulate production processes, easily, we have to generate code from such statecharts. Our code generation approach for statecharts is inspired by the state-design pattern, cf. [GHJV95] and by [AT98] and by the Rhapsody case tool [HG96, Rhap]. However, these approaches generate specific new classes for each state employed in the state-chart. This results in the generation of a complicated, large inheritance hierarchy of many specific, small state classes that mainly redefine a small number of methods, only. [Doug98, chapter 6.2.3] uses a generic array based state-table. This idea was attracting to us, however, we considered the array based implementation as too inflexible.

Our approach adapts the idea of [Doug98] but uses an object structure to represent the state-table at runtime, cf. [KNNZ99]. We use objects to represent the states of a statechart and attributes to hold the do-, entry-, and exit-methods. These state objects are linked via transition objects that store the triggering events and possible guard and action methods attached to a transition. Additional links and attributes represent e.g. the nesting of complex states and identify initial states, history states, deferred events, etc. Each reactive object (e.g. each shuttle) holds its own runtime representation of its statechart. In addition, we provide a library function handleEvent that is able to interprete the state-table and to react on events as specified in the statechart and to issue the appropriate action methods and to switch to the resulting states.

Using a generic state-table representation at runtime, all code that needs to be generated from a statechart by the

Fujaba environment is a single method initStatechart per reactive class. Method initStatechart is called within the constructor of the reactive object. It builds up the state-table, attaches it to the reactive object and starts an event handling thread. For the event methods of a process class we generate a simple standard implementation. The event methods create an event object encapsulating the event name and possible parameter values. Then, the event object is enqueued to the event queue where it is picked up by the event handling thread. The event handling thread handles the event according to the current state and the state-table structure. Finally, auxiliary methods encapsulating transition guards and the action methods have to be created.

Note, to send an event to a reactive object one just calls the according event method. This provides a very convinient and uniform way to issue services on an reactive object. It also enables a standard adaption of inter process communication services like Java-RMI or CORBA.

In our approach, actions used in statecharts may perform complex computations that may need some time. For example, the go activity used in Figure 6 does a shortest path computation to minimize traveling times. To deal with these computation times, our statechart implementation uses event-queues and handles events asynchronously. Internal events (like event reached within class Shuttle) create and enqueue event objects, too. These internal event objects gain a higher priority than external events. This ensures that all reactions to an external events (direct activties and secondary steps caused by internal events) are completed before the next external event is handled. Note, our statechart framework implements a so-called run-to-completion semantics. For a more complete discussion of these issues see [KNNZ99, Köhl99].

Note, in our approach all events are explicity targeted to their receivers. (One actually calls a method on the receiving object.) Targeting events explicitly models communication channels between production agents as specified in the SDL block diagrams. Targeted events have already been introduced by UML statecharts. However, usually statecharts assume a broadcast mechanism distributing events to all available reactive objects. To support this style of event handling, our framework provides a class FBroadcaster. In case of broadcast events, the application creates an implicit broadcaster object. All reactive objects (interested in these kinds of events) register themselfs at the broadcaster for the events they are interested in. Broadcast events are created as usual and then pushed into the event queue of the broadcaster. The broadcaster handles the received events by forwarding them to the event queues of the reactive objects that registered their interest in this kind of events. Note, an application may employ additional broadcasters, e.g. in order to establish broadcasting of certain events for certain groups of reactive objects. Such group events are just send to the group broadcaster object.

Using more than one event queue allows a more concurrent handling of events using multiple threads. However, this probably changes the 'order' in which events are consumed and thus has semantic relevance. In addition, the usual

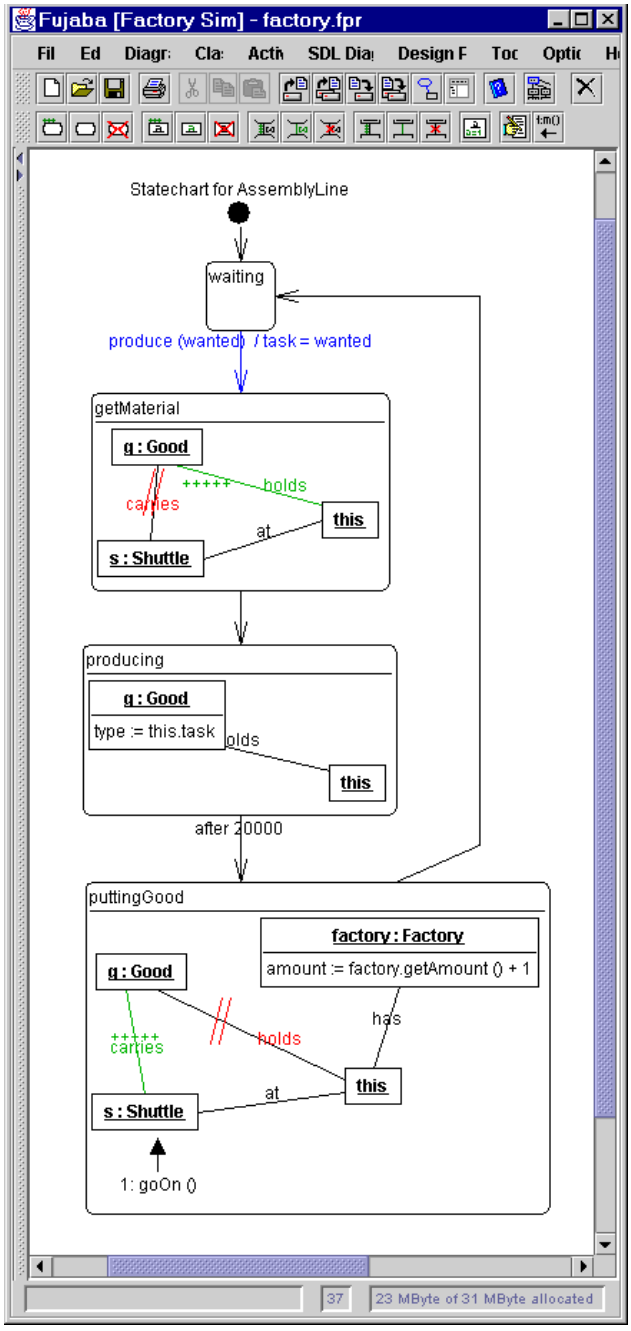

Figure 7 Story-Chart for class AssemblyLine

problems attached to concurrent executions, like race conditions and deadlocks, are raised. Of course, a sound specification should avoid such problems. Static checking for (recognizable yet frequently occuring) specification errors raising these problems is current work.

7 COMBINING STATECHARTS AND COLLABORATION DIAGRAMS

The previous chapter described how one may use statecharts to model the reaction of a process object to signals. Depending on their current state, process objects execute certain actions and change to new states. What is missing are appropriate means for the *modeling of the actual actions*

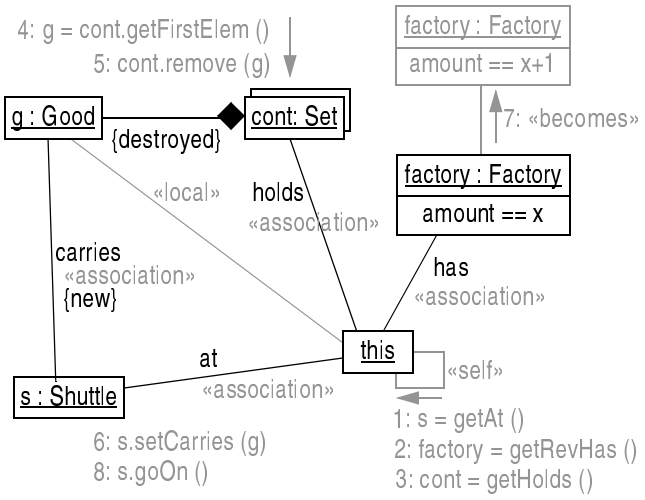

Figure 8 Detailed story-diagram for action doPuttingGood

triggered by the signals. Therefore, we propose to combine statecharts and collaboration diagrams. We use statecharts to specify complex control flows and collaboration diagrams to specify the entry, exit, do, and transition actions that deal with complex object-structures.

Originally, collaboration diagrams are intended to model scenarios of complex message flows between a group of collaborating objects. In this context, collaboration diagrams do not have a precise execution semantics. In order to use collaboration diagrams as a visual programming language one has to add a lot of details on how the participating objects are found and how object structure modifications are executed. As an example, Figure 8 shows a detailed collaboration diagram for the activity doPuttingGood which is part of the statechart for class AssemblyLine, cf. Figure 7.

The collaboration diagram of Figure 8 employs 5 objects that are connected via various links. The collaboration messages 1 to 3 look up associations attached to the this object and fill variables s, factory, and cont with appropriate values. Note, that according to the class diagram, holds is an to-many association. Thus, looking up the holds association results in a set of objects represented by the multi-object cont. Step 4 retrieves the first element from this set of objects and assigns the result to variable g. At this point, all participating objects are found and the actual operations can be executed. Step 5 removes the retrieved good g from the set cont and step 6 creates a carries link between shuttle s and good g and step 7 changes the value x of the amount attribute of object factory to the new value x+1. Finally, step 8 sends a goOn signal to shuttle s.

Provided with such detailed collaboration messages code generation becomes very simple, cf. Figure 9. However, this usage of collaboration diagrams adds little abstraction compared to usual pseudo code. We raise the level of abstraction by assigning a standard semantics to certain graphical elements of collaboration diagrams and by a systematic simplification of required elements. For the collaboration diagram of Figure 8 this will result in the simplified collaboration diagram shown as do action of state puttingGood at the bottom of Figure 7.

In our context, a collaboration diagram specifies the body of a certain do method. At this, all collaboration messages originate from the this object. All boxes depict local variables of the current method or globals declared in the class diagram. Thus, it is clear which kind of objects are denoted and all «self», «global», and «local» links may be omitted. Finally, only «association» links will remain. Thus, the «association» stereotype is superfluous, too. Next, it is clear, that the method variables have to be filled using the depicted links between the objects. The Fujaba code generator is able to compute the necessary look-up operations, automatically, cf. [Zün96b, FNTZ98]. Thus, one may omit the association look-up operations (step 1 to 4). In Figure 8, the multi-object cont is used to look up the holds association and to select one of its elements. Our code generator derives these two steps, automatically. Thus, a single holds link from this to g suffices. Next, the depicted constraints {new} and {destroyed} indicate the creation and deletion of the corresponding elements, clearly. Thus, steps 5 and 6 can be generated by our code generator and may be omitted. Additionally, we simplify the frequent case of attribute assignments by allowing inscriptions like amount:=amount+1 in the attribute compartment of an object. Thus, the two occurences of the factory object and their connecting «becomes» link may be combined to a single box.

The bottom state in Figure 7 shows the resulting simplified collaboration diagram for activity doPuttingGood. In addition to the discussed simplifications we use series of (green/light grey) plus symbols and two parallel (red/dark grey) cancelling lines instead of {new} and {destroyed} constraints, respectively. Experiences in teaching our notation to students have shown, that the textual constraints are frequently overlooked and that the new graphical notation is much easier to read.

In our usage, collaboration diagrams depict the effects of operations in terms of changed attribute values and created and destroyed objects and links. Thus, the initial situation modeled by a collaboration diagram corresponds to the left-hand side of a graph rewrite rule. Accordingly, the situation resulting from the execution of the collaboration diagram corresponds to the right-hand side of that graph rewrite rule. This view allows the execution and translation of collaboration diagrams using code generation techniques known from the graph grammar field, cf. [SWZ95, Zün96, FNTZ98]. In addition, the rich graph grammar theory facilitates the proof of complex system properties, cf. [Roz97] for an overview of graph grammar theory and [JZ99] for an application of this theory to the database re-engineering field.

Figure 7 shows how we combine statecharts and collaboration diagrams to so-called *story charts*. Figure 7 shows the story chart of our assembly lines. Our assembly lines loop through 4 major states. Usually an assembly line is in state waiting. When it receives a produce event, it stores the wanted parameter into its task attribute and changes to state getMaterial. State getMaterial shows its do method as a

```
 1: public class AssemblyLine ... { ...
 2:    void doPuttingGood () {
 3:       Shuttle s; Factory factory;
 4:       Set cont; Good g; int x;
 5:       s = getAt ();
 6:       factory = getRevHas ();
 7:       cont = getHolds ();
 8:       g = cont.getFirstElem ();
 9:       cont.remove (g);
10:       s.setCarries (g);
11:       x = amount;
12:       amount = x+1;
13:       s.goOn ();
14:    } // doPuttingGood
15: } // AssemblyLine
```

Figure 9 Java Code for action doPuttingGood

collaboration diagram, directly. The do method looks up the shuttle s that should currently stay at this assembly line and the good g that is carried by s. It just cancels the carries link between g and s and creates a holds link between this and g, thus simulating that the assembly line takes the carried material from the shuttle. The collaboration diagram of the next state producing simulates that the assembly line turns the held material into the desired good. For simplicity reasons this is done by simply changing the type attribute of g to the value of this.task. The transition to state puttingGood fires after 20000 milliseconds, simulating that the manufacturing process needs some time. Finally, state puttingGood loads the manufactured good on the shuttle again and increments the amount of manufactured goods of its factory. In addition the collaboration message 1: goOn() sends a signal to the shuttle which should then switch from its state produce to state goDeliver and proceed with its tasks, cf. Figure 6.

Figure 7 illustrates the expressive power of story charts, the combination of statecharts and collaboration diagrams. Beyond just specifying the reactions to signals in terms of method activations and state changes, the embedded collaboration diagrams show the object-structures that model the concrete internal states of our production agents and how these states evolve over time. Due to our experiences with a combination of activity diagrams and collaboration diagrams cf. [JZ98, FNTZ98, NZ99] the combination of high-level control flow specifications (statecharts or activity diagrams) and high-level object structure rewrite rules (collaboration diagrams) results in a powerful visual progamming language. This powerful visual progamming language is an ideal means for the specification of the reactive behavior of flexible production agents that use an object-oriented representation of the world they are living in and of the plans and tasks they are executing. Note, due to the formal semantics of statecharts and graph rewrite rules, the semantics of story charts is well defined. Similarly, the Fujaba environment provides a code generator for story charts generating a table-driven implementation for the

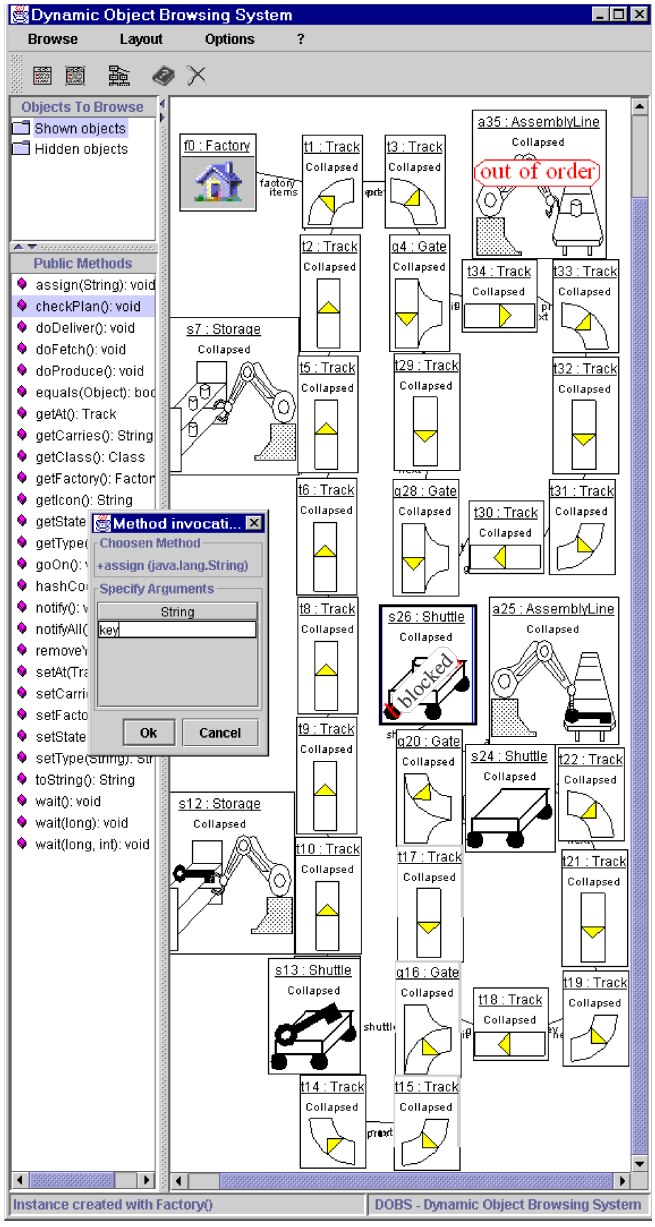

Figure 10 Simulation of a simple production system

statechart parts and usual Java code for the collaboration diagrams as shown in Figure 9.

8 VALIDATION VIA SIMULATION

Once the specification reaches a complete and consistent state, the Fujaba *code generators* may translate them to an executable Java implementation. To facilitate *simulation* and graphical debugging of the generated application, the Fujaba environment provides *Mr. Dobs*, our *D*ynamic *O*bject *B*rowsing *S*ystem. Mr. Dobs uses Java runtime type information and dynamic method invocation features for analysing Java runtime object structures and for the user driven execution of application specific methods.

Figure 10 shows a screen-shot of a Dobs session running our example specification. The depicted topology is derived from Figure 3. Note, on the creation of reactive objects, like

shuttles, stations, and gates, the corresponding event handling threads start automatically. After selection of an object, the left column of Mr. Dobs shows all invokable methods of that object. One may interactively invoke one of these methods. If the invoked method requires parameters, a generic dialog window is opened, allowing to enter parameter values. After a(n interactive or event driven) method execution, Mr. Dobs reflects the changed object structure. In Figure 10 we have invoked the event method assign for all shuttles, asking them to produce keys and locks. From that on, the shuttles execute their manufacturing task as specified in Figure 6. They travel along the tracks, comunicate with gates, assembly lines, and storages, fetch material, trigger assembly lines, and deliver the created goods. Studying the running simulation, one may already observe specification errors like communication problems or deadlocks. One may also recognize bottlenecks or productivity problems due to unemployed production agents. Within the running system, one may create new objects, e.g. additional gates and tracks, and connect them to the existing object structure. Thereby, alternative system configurations might be tested. Additionally, one may simulate defects of some components, e.g. by disconnecting an assembly line, and analyse the behavior of the remaining production agents. In Figure 10 we just have disabled the upper-right assembly line. Due to their flexible specification, our shuttles are not threatened by this drop-out, but just travel to the alternative lower-right assembly line. However, the single functioning assembly line has become a bottleneck such that one shuttle (left of the lower assembly line) is already blocked, queuing for service.

Mr. Dobs already provides an easy to use but restricted animation possibility for the Java code generated from story-chart specifications. A better integration with the specification, e.g. an animation of the executed story-charts for selected reactive objects is current work. We also try to improve the user interaction e.g. by attaching buttons and other GUI elements to depicted objects. Eventually, an adapted version of such an user interface might even serve as an user interface for the actual production system.

9 CONCLUSIONS

This paper discusses the use of UML diagrams as a visual programming language. For class diagrams the translation to an object-oriented program is straight-forward. The translation of statecharts is based on a table driven approach inspired by [Doug98]. The use of collaboration diagrams and their translation to Java is taken from our previous work [FNTZ98, JZ98, ZSW99]. The main contribution of this paper is the integration of SDL block diagrams and class diagrams and statecharts and collaboration diagrams to a sound and complete specification language for flexible production control systems, as discussed in chapter 3 and shown in Figure 2. The resulting specification language and modeling approach combines the power of statecharts, providing sophisticated means for modeling concurrent reactive objects, with the power of graph grammars, providing appropriate means for the specification of application specific object-structures on a very-high level of abstraction.

In addition, this paper describes a possible semantics for the various UML diagrams (via their translation to Java). This may facilitate the reading of certain UML diagrams (by thinking in terms of the corresponding implementation concepts) and clarify ambigous modelings. In addition, one may learn which UML diagram should be used for which purpose and how different UML diagrams may be combined to cover mixed cases. Thereby, we think that one may transfer our results to other application areas.

Currently, the Fujaba environment supports editing of SDL block diagrams and UML class diagrams and story-charts and story-diagrams that combine statecharts and activity diagrams and collaboration diagrams. Code generation is provided for all these diagrams but SDL block diagrams. We already generate class diagrams from SDL block diagrams. The generation of process distribution and initialisation code from SDL block diagrams is current work.

The current prototype of the Fujaba environment is available as free software and comprises about 205 000 lines of pure Java code. The release of Fujaba is available via:

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

Note, so far we simulate multiple concurrent production agents via threads that comunicate via events and share some common memory. The distribution of production agents on multiple processors without a shared memory is current work within the ISILEIT project in collaboration with our mechanical engineering department. This will eventually not only allow the simulation of production agents but also the generation of real production control software that operates the physical actors and sensors of a production system.

In our application domain, we deal with concurrent processes that face the usual concurrency problems like deadlocks, race-conditions, lifeness, safety. Equipped with the rich theory of statecharts and graph grammars, future work will try to provide verification mechanisms for simple cases of these problems.

ACKNOWLEDGEMENTS

The following people contributed substantially to this work with fruitful discussions, careful proof readings and a lot of suggestions: Prof. W. Schäfer, Dr. R. Heckel, J. Wadsack. Thank you very much.

REFERENCES

[AT98] J. Ali, J. Tanaka: *Implementation of the Dynamic Behavior of Object Oriented System*; IDPT Vol. 4, 19998, Proc. of third biennial world conference on integrated desgin and process technology, 281-288, ISSN No. 1090-9389, Society for Design and Process Science (1998)

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

[Doug98] B. P. Douglass: *Real Time UML*; Addison Wesley, ISBN *0-201-32579-9 (1998)*

[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)

[GHJV95]E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*; Addison Wesley, ISBN 0-201-63361-2, 1995

[GK97] U. Glässer, R. Karges, Abstract State Machine Semantics of SDL, in Journal of Universal Computer Science, Vol. 3, No., 12, 1997

[H+88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Tauring, and M. Trakhtenbrot, STATEMATE: A Working Environment for the Development of Complex Reactive Systems, IEEE Trans. Soft. Eng., 16, 403-414, 1990,

[HG96] D. Harel, E. Gery: *Executable Object Modeling with Statecharts*; Proc. 18th Int. Conf. on Software Engineering (ICSE '18), Berlin, pp 246-257, IEEE, SIGSOFT, ISBN 0-8186-7246-3 (1996)

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

[JFC99] Technical reference of the Java Foundation Classes, contained in the Java Development Kid v1.2 (Java2) http://www.sun.java.com/

[JZ98] J.-H. Jahnke and A. Zündorf: *Specification and Implementation of a Distributed Planning and Information System for Courses based on Story Driven Modelling*; in proceedings of the Ninth International Workshop on Software Specification and Design April 16-18, Ise-Shima, Japan, IEEE Computer Society, pp. 77-86, ISBN 0-8186-8439-9

[JZ99] J.-H. Jahnke and A. Zündorf: Applying Graph Transformations To Database Re-Engineering. Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2 - Application. World Scientific, Singapore, 1999. To appear.

[NZ99] J. Niere, A. Zündorf: Using Fujaba for the Development of Production Control Systems; accepted for Proc. of AGTIVE 99 - Applications of Graph Transformations With Industrial Relevance, Monastry Rolduc, Kerkrade, The Netherlands, Sept. 1-3, 1999, LNCS, to appear, 1999

[KNNZ99]H. J. Köhler, U. Nickel, J. Niere, A. Zündorf: *Using UML as a visual programming language*, technical report, tr-ri-99-205, University of Paderborn, 1999

[Köhl99] H. J. Köhler: *Code Generation for UML Collaboration, Sequence, and* Statechart Diagrams; Master Thesis, in German, Dep. Computer Science, University of Paderborn, 1999.

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

[Roz97] G. Rozenberg (ed): Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific, 1997.

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

[San99] T. Sander: Tool Support for the Design and Generation of Agent Systems, in German, Master Thesis, University of Paderborn, 1999

[SGW94] B. Selic, G. Gullekson, P.Ward, Real-Time Object-Oriented Modeling, John Wiley & Sons, New York, NY, 1994.

[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.

[Zün96] A. Zündorf: A Development Environment for PROgrammed Graph REwriting Systems; (in German), Dissertation, RWTH Aachen, Germany, 1996.

[Zün96b] Zündorf A.: Graph Pattern Matching in PROGRES; In: Cuny J., Ehrig H., Engels G., Rozenberg G. (eds.): Proc. 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science; LNCS 1073, Springer, 1996.