Using Fujaba for the Development of Production Control Systems

Jörg Niere, Albert Zündorf

AG-Softwaretechnik
University of Paderborn, Germany
[nierej|zuendorf]@uni-paderborn.de
D-33095 Paderborn

Abstract. Current production control systems for e.g. a factory for cars or any other complex industrial good face two major problems. First, production control systems need to become (more) dezentralized to increase their availability. It is no longer 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. Common specification languages for embedded systems, like SDL, statecharts, etc. 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 (more) intelligent, autonomous production agents. Such 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. This paper proposes to use (object-oriented) graph structures for the representation of production agents and graph (object structure) rewrite rules for the specification of their behaviour. We show how the FUJABA environment may be used to specify production agents and generate their implementation and to validate them via a graphical simulation.

1 Introduction

In [1] Blostein states that practically applicable graph transformation systems should allow to combine graph transformations and conventional (object-oriented) programming code, seamlessly. In order to achieve this, *Fujaba*[1] combines common object-oriented notations, i.e. UML class diagrams, UML activity diagrams, and UML

1. **F**rom **U**ML to **J**ava **A**nd **B**ack **A**gain

collaboration diagrams, into a visual specification language that integrates object-oriented modeling, normal (Java) code, and graph transformations ([2], [3], [8]).

Like other CASE tools, the *Fujaba* environment allows to edit UML class diagrams and provides a code generator that generates Java classes containing attributes and method declarations. In addition, the Fujaba environment generates canonical implementations for associations. To specify method bodies, Fujaba uses so-called *story diagrams*. Story diagrams combine UML activity diagrams and UML collaboration diagrams. Activity diagrams define the high-level control flow between different activities. The activities may be specified either by normal Java code or by a graph transformation depicted as a UML collaboration diagram. The control flow specified by an activity diagram is translated into standard Java while and if statements (see [6]). Activities containing standard Java code are just copied into the resulting control structures. The graph transformation / collaboration diagrams are translated into Java code that relies on the class features generated from the class diagram, only. This is done using query optimization techniques adapted from the database field and refined for graph transformations ([2], [12]). The resulting code employs usual main memory object structures to represent the host graph. It does not rely on special graph libraries or graph rewrite rule interpreters but it manipulates the object structures by usual pointer operations. Thus, the resulting code blends seamlessly with other system parts and is not resource demanding. Finally, it is 100% pure Java code, that runs on any Java platform. Altogether, this enables the application of graph rewriting techniques for various new areas, e.g. for embedded systems.

As part of the ISILEIT project funded by the German Research Society (DFG) our department studies the application of formal methods to embedded systems. Actually, the running example of this paper stems from the reference case study of the ISILEIT project [8]. The ISILEIT project is a joined project with electrical and mechanical engineers. The focus is not just on theoretical results, but there are also strong demands for practical benefits, like e.g. the generation of running production systems from their formal specification and the reduction of system reconfiguration times. Thus, we propose to attack these problems with Fujaba.

Embedded systems are not yet a well known application area of graph rewriting systems. Common specification languages for embedded systems, like SDL, statecharts, etc., 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. However, these common specification languages provide no appropriate means for the specification of (more) intelligent, autonomous production agents. Such autonomous production agents need knowledge about manufacturing plans for different goods and of their surrounding world, i.e. the layout of the factory and the availability of material and manufacturing cells. In addition, such production agents have to coordinate their access to assembly lines with other competing agents. Finally, production agents should deal with unforseen situations, like the drop-out of a production line or changes to the production plans.

Altogether the requirements for intelligent production agents are close to general process modeling requirements. Fortunately, there exists already a sophisticated graph grammar based approach to process modeling, namely the Dynamite project [5]. Thus, our approach has taken the dynamic task net idea from the Dynamite project and adapted these task nets for the needs of intelligent production agents and uses Fujaba to specify and implement these nets.

In the following chapter a short introduction of a simple production system example is given. Chapter 3 describes the class diagram and the activity diagram of a method. In chapter 4 a cut-out of the generated Java code for the specified method is presented. The following chapter introduces the test environment of Fujaba. The last chapter gives conclusions and future work issues.

2 Sample Factory Example

Figure 1 shows a scenario of a sample factory used as running example within the paper. The example stems from a real world system that serves as the case study for our ISILEIT project funded by the German Research Society (DFG) [8]. The factory is modeled as a flat building without levels and pillars in it. The floor is layered with rectangle shaped fields allowing to address certain positions in the building. The factory contains certain kinds of production places. A production place is e.g. an assembly line, where goods arrive and are loaded on shuttles.

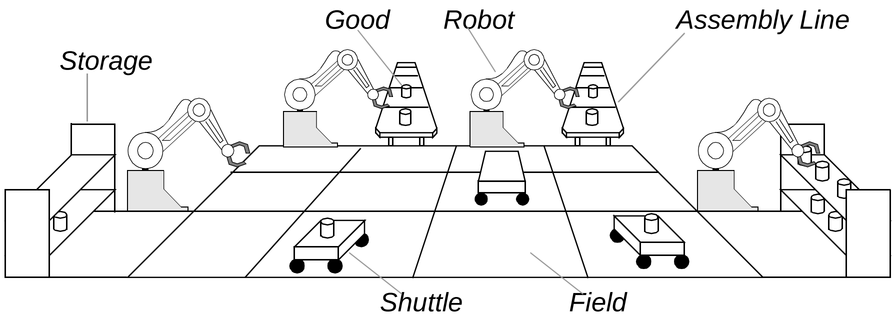

Figure 1 Simple factory example

Shuttles are able to move over the floor, where the fields on the floor serve as a map. Each field can be allocated with only one shuttle, so a shuttle must be able to dodge a field which is allocated by another shuttle. Shuttle accept orders to carry goods from an assembly line to a storage. Thereby, shuttles can only carry one good and orders will be executed until there are no more goods on the assembly line, the storage is full or the shuttle receives a new order.

In order to meet these requirements and to force the autonomy of shuttles, a working plan should be specified for example with a task net. The net may look like the one shown in Figure 2. Each shuttle can be assigned to a working plan (here produce_good) and will then execute the actions in the working plan autonomously. Tasks may require certain resources like task mill_cutting, which will require an assembly line later on. The task net is hierarchically organized, e.g. task produce_good has the subtasks fetch_material, mill_cutting, and deliver. The tasks are conneted by next links in order to specify a sequence of execution. Some tasks may have more than one successor task,

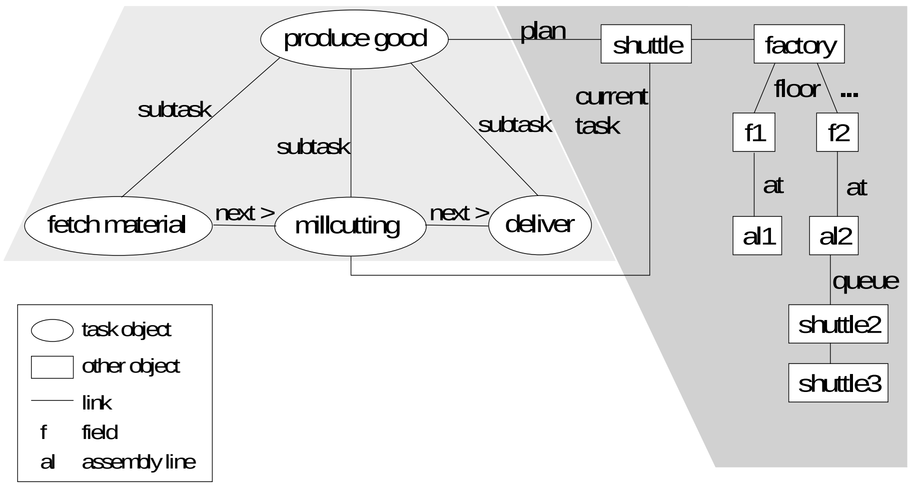

Figure 2 Task net for shuttles

modeling independend steps that may be executed in any order. Similarily, one task may require multiple predecessor tasks to be completed until it becomes executable. Based on this task net idea, the following chapter outlines the static aspects of our example.

3 Class and Story Diagrams

Based on the requirements and motivations outlined above, we propose the following sample design for intelligent production agents, cf. Figure 3. Class Shuttle models transport vehicles which are our production agents. These production agents need to know about the Factory they are living in and about the AssemblyLines and Storages they are negotiating with and about Goods to be produced. Therefore, class Actor has a actors association to the factory. Nevertheless, class Actor is a design decision we made for this example and encapsulates common parts of other classes, only. Next, production agents need to be aware of their own location and of the locations of assembly lines and storages. This is achieved via class Field. Each field has a certain position within the factory stored in its attributes x and y. Links of type horizontal and vertical allow shuttles to move from a field to its neighbors. A target link identifies the current movement target and at links model the current locations of all Actors. Next, class Task models the working plan of a production agent as described above. We use subtasks links to structure hierarchical plans and next links to represent the successor relationship. Finally, tasks may allocate assembly lines or storages via resource links. Note, this is a very simple modeling of work plans for simplification reasons. More sophisticated versions could e.g. model priorities, durations, and other pertchart properties, in addition. The root of a production plan is attached to its executing production agent via a plan link. In addition, a link named current identifies the active task. So far a production agent has knowledge about the configuration of the factory it is living in. To allow multiple production agents to coordinate themselves, they need to know about each other and about their plans. Actually, the class diagram described so

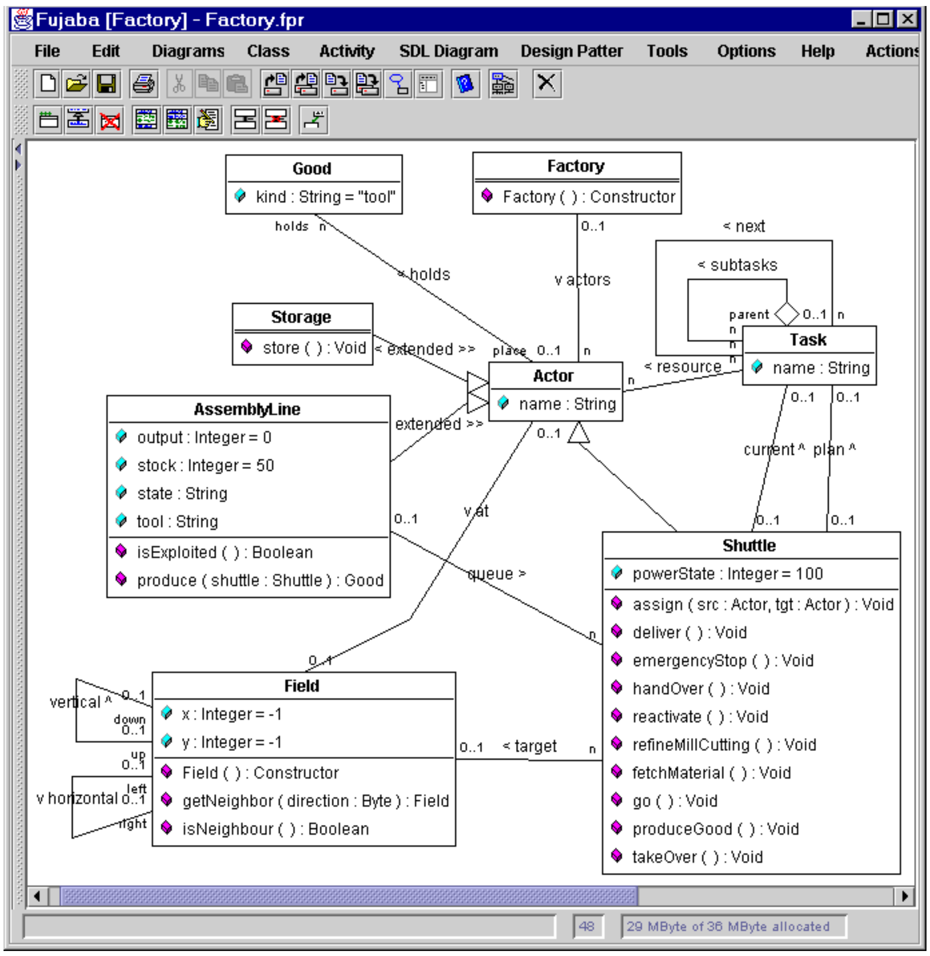

Figure 3 Class diagram of the factory example

far allows to represent such information within each production agent. However, to minimize communication efforts, shuttle coordination is restricted to waiting queues at the different assembly lines. When planning a manufacturing task, a production agent may look up the waiting queues of assembly lines in order to choose one with minimal waiting time.

Equiped with this static design, we are now able to specify the behavior of our production agents using story diagrams. Story diagrams combine UML activity diagrams and graph rewrite rules. Therefore, UML activity diagrams contain activities and directed transitions among activities to specify the (high-level) control flow. In UML, activities can contain "pseudo" code, only. Story diagrams extend the notation by using either Java code or graph rewrite rules for the specification of activities. Special transition guards allow to branch on the success or failure of a graph rewriting step.

Our production agents shall not just execute a fixed plan (which could actually just have been hard coded) but they should refine their production plans depending on different situations like e.g. the availability of assembly lines. As a simplified example for such a flexible planning step, Figure 4 shows a story diagram used to plan a millcutting step. Fujaba uses a notation for graph rewrite rules that is similar to UML collaboration diagrams. In this notation, the left- and right-hand sides of a rewrite rule are displayed in one picture: elements to be deleted are canceled with two (red) lines and newly created elements are marked by attached (green) plusses (see [2], [3] for more details). Such a graph rewrite rule is executed by first finding a match for the variables and links in the rewrite rule to objects and links in the runtime object-structure and then executing the depicted modifications. Thus, the example operation millCutting looks up its factory and seeks for an assembly line al that (1) is in state active, (2) that currently operates a millcutter, and (3) that has no other shuttle waiting in its queue. In addition, we look-up field f that identifies al's position. On success, operation millCutting creates the three new subtasks go, hand_over, and take_over for the millcutting step. Field f is marked as the shuttle's next movement target and the shuttle enqueues itself at the assembly line. Finally, tasks hand_over and take_over mark the chosen assembly line al as their resources. Note, that Figure 4 shows only the first task refinement attempt. When the depicted graph rewrite rule is not applicable, execution proceeds along the [failure] transition and considers less optimal plan refinements. Otherwise, we proceed along the [success] transition and operation millCutting terminates and execute the new current go task.

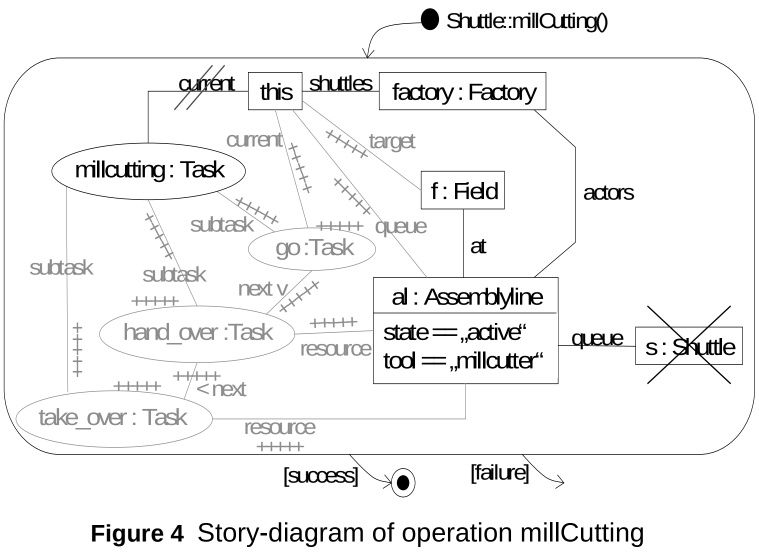

Figure 4 Story-diagram of operation millCutting

For each kind of task, such an execution routine is provided. Plan execution starts at the plan root. It chooses an executable task and calls the corresponding execution routine and marks the taks as done. In addition, our model allows to suspend currently executed tasks in order to switch to another executable task e.g. for optimization reasons or in order to react on unforseen difficulties like e.g. the drop-out of an assembly line. Once a plan is fully executed, it is replaced by a new one and execution starts again. In addition, a shuttle might keep some approved plan parts in order to retain successful execution pathes.

Note, in addition to the shown plan execution operations that model the knowledge of our production agents and their manufacturing strategies, shuttles also need to control their sensors and actors like light bars and their motors and they have to react on signals sent from other agents or from humans, e.g. assigning them new tasks. This reactive behavior is well addressed using SDL and statecharts. Thus, Fujaba provides support for SDL and statecharts, too. This topic is addressed in the Fujaba tool demonstration description, which is part of this volume, too [10].

4 Java Code Generation

FUJABA provides a generator, which generates Java code out of a specification. For each class in a class diagram the corresponding Java class is generated in a canonical way. Attributes are encapsulated and access methods are generated. Method declarations are mapped directly into corresponding Java method declarations and associations are mapped to pairs of references and adequate access methods within the corresponding classes (for more details see [2], [3]).

The Java code generation for story diagrams is divided into two tasks. First, the control flow is mapped to imperative control structures like if, and while statements. To enable this translation, story diagrams are restricted to so-called well-formed transition structures that correspond directly to nested branches and loops.

```
public void millCutting () { ...
// first graph rewriting rule
try
{
    sdmSuccess = false;
    millCutting = this.getCurrent ();
    SDM.ensure (millCutting != null);
    factory = this.getFactory();
    SDM.ensure (factory != null);
    Iterator actors = factory.iteratorOfActors ();
    while ( ! sdmSuccess && actors.hasMore ()) {
        tmp = actors.next ();
        try {
            SDM.ensure (tmp instanceof AssemblyLine);
            al = (AssemblyLine) tmp;
            SDM.ensure (al.getState () == "active");
            SDM.ensure (al.getTool () == "millcutter);
            SDM.ensure (al.queueIsEmpty ());
            f = al.getAt ();
            SDM.ensure (f != null);
            // match found, execute modifications
            this.setCurrent (null);
            goto = new Task ("goto");
            handOver = new Task ("handOver");
            takeOver = new Task ("takeOver");
            this.setCurrent (
            this.setTarget (f);
            al.addToQueue (this);
            millCutting.addToSubTasks (goto);
            millCutting.addToSubTasks (handOver);
            millCutting.addToSubTasks (takeOver);
            goto.addToNext (handOver);
            handOver.addToNext (takeOver);
            handOver.addToResources (al);
            takeOver.addToResources (al);
            sdmSuccess = true;
        } catch (SDM.Exception sdmExcept) { }
    } // while (actors.hasMore ())
} catch (SDM.Exception sdmExcept) { }
if (sdmSuccess)
{
    return;
} else
{ // next graph rewrite rule
        . . .
```

Figure 5 Java code for graph rewrite rules

Figure 5 shows the Java implementation of the millCutting method of class Shuttle. Lines 2 to 39 implement the first graph rewrite rule shown in Figure 4. The if statement in line 40 realizes the control flow depicted by the success and failure transitions at the bottom of Figure 4. If the first graph rewrite rule was successful, the first if-branch terminates the execution. Otherwise, the else branch is executed.

In a second task, the code for activities is generated. Activities, that contain just Java code are copied one-to-one. For graph rewrite rules we employ translation mechanisms as described in [2], [12]. In Figure 5, lines 2 to 39 show the generated Java code for the first graph rewrite rule. The execution starts with binding objects to the variables specified in the rule. For example, in line 6 the variable millCutting is bound to an object which is accessable via an current link from the this object. Line 7 checks whether line 6 actually retrieved an object and throws an exception, otherwise. This exception is caught within the catch-statement at line 39. Note, variable sdmSuccess is set to false in line 5 and thus it signals that the execution of the first graph rewrite rule has failed until it is set to true in line 36. Line 36 is reached only when all SDM.ensure clauses are passed, successfully. While line 6 looks-up a to-one association, the link from variable this to variable al belongs to a to-many assoctiation, cf. Figure 3. Thus, we need a loop (line 10 to 12) to look up all reachable neighbors until we reach one that meets all requirements: we are looking for an assembly line (line 14) which is active (line 16) and which operates a millcutter (line 17) and which has an empty queue (line 18). Once all participants are identified, we execute the deletions (line 22) and create new objects and links (line 23 to 35) and finally we signal success of the rewrite step (line 36). Note, that the latter aborts the while loop in line 11. See [3] for more details on the code generation for graph rewrite rules.

The important properties of the generated Java code are, that it operates on usual main memory object structures and that it uses only small library functions like predefined container classes and the rule execution is programmed built-in, it does not rely on an additional rule interpreter. Thus, the resulting code is not very resource demanding. In addition, it is 100% pure platform independant Java code that does not use any native methods. Altogether, these features enable us to use Fujaba for the generation code for embedded systems.

5 Simulating the specification

Our approach to specify production control systems already allows to construct very flexible production agents that allow very small lot sizes and that may manufacture different goods in parallel. These production agents are able to deal with unforseen situations, like assembly line drop-outs. They form a decentralized control system that is not threatened by the drop-out of a single central production control computer. Still, we have to meet the requirement of being able to switch to new products without long down-times caused by system tests. To meet this requirement, we propose to test production processes beforehand with Fujaba’s graphical debugging and simulation environment, called DOBS (Dynamic Object Browsing System), cf. Figure 6. The DOBS environment allows to visualize (Java) runtime object structures and to invoke methods on objects, interactively. For parameterized methods, appropriate user dialogs

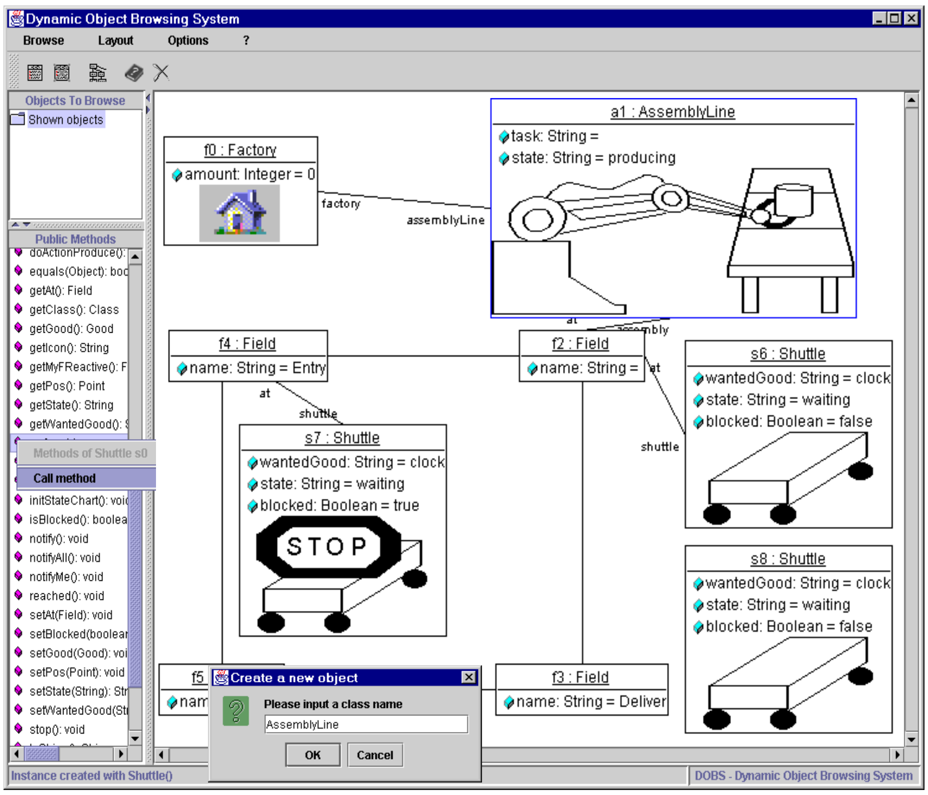

Figure 6 Simulating the sample factory example

are created, dynamically. In addition, DOBS is able to deal with the (re)active objects generated by Fujaba, that run their own thread and thus may change their state or execute methods, autonomously. Thereby, DOBS may serve as a first simple graphical user interface allowing to test a story diagram specification. For example, the user may initiate a production process and then e.g. simulate the drop-out of a certain assembly line and analyse the reaction of the running production agents. On the contrary, during the simulation one might recognize some bottlenecks and try to solve the problem by adding more assembly lines or shuttles, interactively. Another option is to reshape the floor layout or to move some assembly lines around in order to shorten distances. Deletion and addition of floor elements might also be used to simulate walls, doors, and pillars. Finally, one might even "edit" the task net of some production agents in order to test alternative production plans.

6 Conclusions and Future Work

This paper shows the applicability of graph rewrite techniques within the area of embedded systems. Due to our experiences within the ISILEIT project, graph rewrite rules are an ideal means for the specification of the general behavior of flexible production agents. The high level of abstration provided by graph rewrite systems allows us to model the 'world' our production agents are living in and the knowledge they need to execute their tasks, very easily. The experiences drawn from the Dynamite project enabled us to provide our production agents with very flexible manufacturing plans. However, to turn embedded systems into an application area for graph rewrite systems, several properties of the Fujaba approach were very important. First, the UML like notation employed by Fujaba facilitated the communication of story diagram specifications to the other ISILEIT project partners, significantly. Note, a number of these partners stem from fields like electrical and mechanical engineering. Second, todays embedded systems have certain resource restrictions. This topic is well addressed by the Fujaba code generation strategies, which produce simple Java code using usual main-memory object structures. We expect that Java will become availabe for wide spectrums of embedded systems, soon. Then, the 100 % pure Java code generated by Fujaba will be executable on the quite heterogenous hardware platforms employed in the area of embedded systems.

One unexpected advantage of graph rewrite techniques and of the code generation strategies of Fujaba is their quite defensive programming style. As Figure 5 shows, our code checks thorougly all kinds of conditions required for the execution of a graph rewrite step by using numerous SDM.ensure clauses. This results in very reliable code that deals correctly with many kinds of unforeseen situations. During the simulations with our dynamic object browsing system DOBS, our engineering partners were impressed by the robustness of the application. One can delete assembly lines or even fields without causing system crashes of running production agents. It is possible to reconfigure the factory layout while the production agents are active and they still react reasonably. Once a new factory configuration is (partly) established, the production agents easily adapt to the changed setting and continue to produce goods. We hope to be able to transfer this robustness and reliability and flexibility from the simulations to real embedded systems. This is current work.

FUJABA has been developed since November 1997. The current 'release' version provides editors for UML class diagrams, UML activity diagrams and object structure rewrite rules. In addition it comprises a code generator and a basic consistency analyser. As current work FUJABA is enhanced by statecharts and SDL. For both languages, first versions of editors, consistency analysers, and code generators are available and in their testing phases, now. DOBS, the Dynamic Object Browsing System, has become part of FUJABA in the beginning of 1998. Extensions of Dobs up to a graphical simulation environment are also current work and are scheduled for 2000.

References

[1] D. Blostein, H. Fahmy, A. Grbavec. *Issues in the Practical Use of Graph Rewriting*. In Proc. 5th Int. Workshop on Graph-Grammars and their Application to Computer Science. LNCS 1073, pp. 38-55, Springer 1996

[2] T. Fischer, J. Niere, L. Torunski. *Design and Implementation of an integrated Development Environment for UML, Java, and Story Driven Modeling*, Master Thesises, Paderborn 1998 (in German)

[3] T. Fischer, J. Niere, L. Torunski, A. Zündorf. *Story Diagrams: A New Graph Rewrite Language based on the Unified Modelling Language and Java*. to appear in Proceedings of TAGT '98 (Theory and Application of Graph Transformations), LNCS, Springer 1999

[4] ISILEIT homepage:
http://www.uni-paderborn.de/fachbereich/AG/schaefer/ag_dt/ISILEIT/index.html

[5] D. Jäger, A. Schleicher, B. Westfechtel, *Using UML for Software Process Modeling*, in O. Nierstrasz, M. Lemoine (eds), Proc of the 7th European Software Engineering Conference, ESEC/FSE '99, September 99, Toulouse, France, LNCS 1687, pp.91-108, 1999

[6] T. Klein, Reconstruction of UML Activity and Collaboration diagrams out of Java Source Code, Master Thesis to appear, Paderborn 1999 (in German)

[7] H.J. Köhler, U. Nickel, J. Niere, A. Zündorf, *Using UML as Visual Programming Language*, to appear as technical report, University Paderborn, 1999

[8] U. Nickel, J. Niere, W. Schäfer, A. Zündorf, *Combining Statecharts and Collaboration Diagrams for the Development of Production Control Systems*, in the Proc. of the OMER Workshop Mai 28-29, 1999 Herrrschingen, Technical Report No. 1999-01, University of German Federal Armed Forces, Munich, 1999

[9] U. Nickel, J. Niere, A. Zündorf, *From UML to Java And Back Again*, Technical Report, University of Paderborn, 1999

[10] J. Niere, A. Zündorf, *Testing and Simulating Production Control Systems Using the Fujaba Environment*, in Proc. AGTIVE Workshop '99, Application of Graph Transformations With Industrial Relevance, Monastry Rolduc, Kerkrade, The Netherlands, Sept. 1-3, LNCS, 1999

[11] G. Rozenberg (ed). *Handbook of Graph Grammars and Computing by Graph Transformation*. World Science, 1997.

[12] A. Zündorf, *Graph Pattern Matching in PROGRES*, In [11], In Proc.5th Int. Workshop on Graph-Grammars and their Application to Computer Science. LNCS 1073, pp. 454-468, Springer 1996