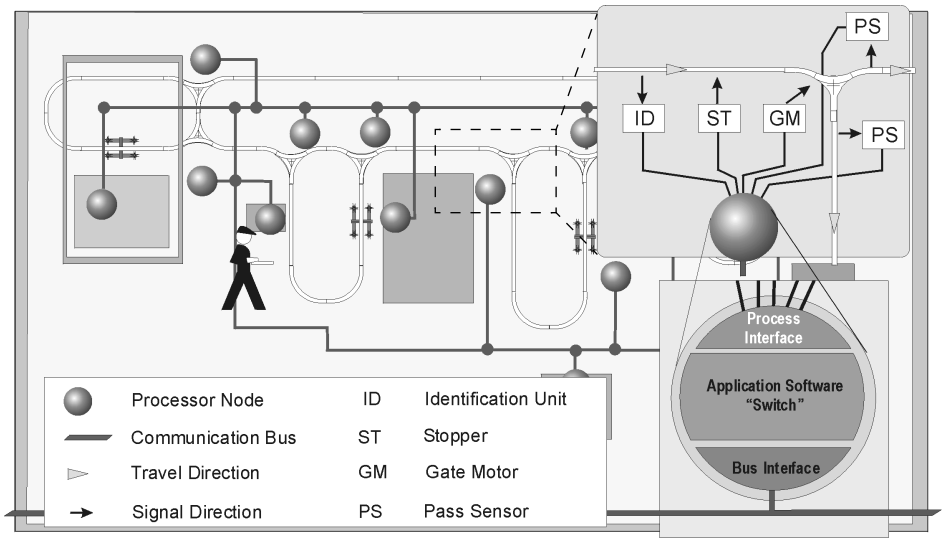

Figure 1 Schematic overview of a material flow system

and actors, e.g. from the identification unit or to the stopper. In general, each node in the network is responsible for its local environment. The control software for the whole material flow system is spread over the nodes and they have to coordinate themselves among each other, to perform the correct task.

The complex tasks of such a system require an efficient way of developing the control software. One of the main problems in todays manufacturing industries is long down times of assembly lines resulting from long testing phases during the installation of the new software on the new hardware. Thus, one wants to validate the specified software beforehand in order to shorten software re-configuration down times of physical assembly lines. In most cases, the verification of the complete system is not possible. The usability of symbolic model checkers for example is not feasible for such complex systems due to the state explosion problem. Hence, simulation environments are employed to validate the system by checking the most important scenarios.

This paper originates from the ISILEIT[1] project which was funded by the German National Science Foundation (DFG). This project aims at the development of a seamless methodology for the integrated design, analysis and validation of distributed production control systems. For this purpose, we have developed a methodology which is a combination of parts of SDL and UML [KNNZ00].

In the following chapter 2, we introduce the architecture of our simulation environment. Chapter 3 describes our running example on which chapter 4 presents the validation opportunities of our environment. We introduce a process-oriented hardware simulation kernel to get a realistic simulation. The kernel is presented in chapter 5. The paper closes with related work, conclusions and future work.

[1] "Integrative Spezifikation von Leitsystemen der flexibel automatisierten Fertigung"
(see http://www.upb.de/cs/isileit for further information)

2 The Simulation Environment Architecture

Figure 2 shows the simulation environment part of our Fujaba[1] environment. In [KNNZ00] we have proposed a modelling approach, supported by Fujaba, for the specification of material flow systems. We propose to start with a so-called topology diagram. Such diagrams represent the 'hardware' configuration of the reference system, e.g. which tracks are connected to each other, and where sensors and actors are placed at the tracks. The topology diagram parameterizes the simulation kernel[2] with the current topology so that the kernel can determine the elements of the material flow system as well as its layout. The kernel executes the simulation model for the considered material flow system.

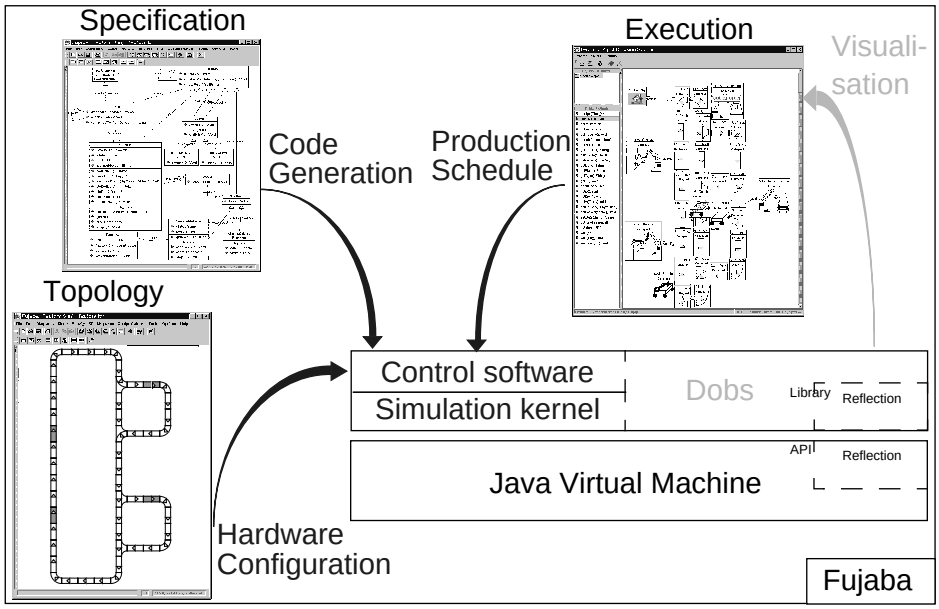

Figure 2 Simulation environment architecture of Fujaba

The control software itself is specified by UML class diagrams and so-called story diagrams provided by the Fujaba environment. In addition, Fujaba is able to generate executable Java source code from a specification. For more details we refer to [FNTZ98, KNNZ00, NZ99]. To observe the running system, we developed the Dynamic Object Browsing system (Dobs) which displays the internal object structure of the Java virtual machine. This includes the control software and the simulation kernel. Thus, one gets an overview of the production sequences which arise by the software controlling the simulation kernel.[3] Since in a real manufacturing system a working plan defines which work pieces have to be produced, we integrated the possibility to define production schedules in Dobs.

---

[1] From UML to Java And Back Again (see *http://www.fujaba.de* for further information)

[2] "A simulation kernel is a program, which provides the modelled world with its elements. It manages the automatic, chronological creation of the events, which are necessary for the correct mapping of a process sequence in the model." [Translated from VDI 3633]

[3] Of course, in a real manufacturing system, a working plan defines which work pieces have to be produced. Hence, we integrated the possibility to define production schedules in Dobs.

3 Running Example

Figure 3 shows the topology of our sample factory used as running example in this paper. The example stems from the ISILEIT project. The topology of our sample factory consists or seven shuttles moving on the track system. Each shuttle executes the same defined working task. Its task starts, when a shuttle is activated, which means that it is assigned to produce a certain good, e.g. locks or keys. 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, the piece of iron is taken from the shuttle, the assigned good is produced, and put on the shuttle again. After that, the shuttle moves to the storage where the good is stored. Finally, the shuttle reaches the end of its working task and starts again from the beginning. The shuttle will perform this task until it gets a new assignment. [KNNZ00] describes in detail the specification of this sample factory and introduces different modelling techniques. These modelling techniques are supported by the Fujaba environment

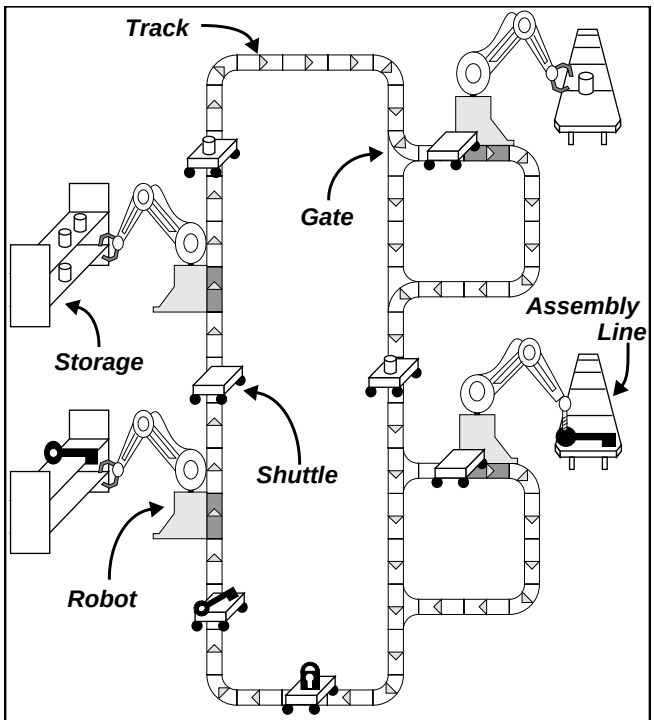

Figure 3 Sample factory topology

4 Validating the system

In contrast to other tools and simulation environments, e.g. STATEMATE [HLN+90], PROGRES [SWZ95], which simulate the specified model like an interpreter, our approach is to generate source code out of the specification and observe the running system. Such an approach closes the gap between the simulation system (interpretation) and the software running on the real system. Our attempt is to use the same code both for the simulation as well as for the running system. So, the generated code has to be free from any kind of de-

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

bug information and could be optimized for special issues, i.e. speed or space optimizations.

To observe the running system, the Fujaba environment provides a graphical debugging tool called ‘Mr. Dobs’ (Dynamic Object Browsing System). Dobs is able to display the internal object structure of a running Java virtual machine. We use original UML object diagrams as graphical representation. Figure 4 shows the running sample factory with the

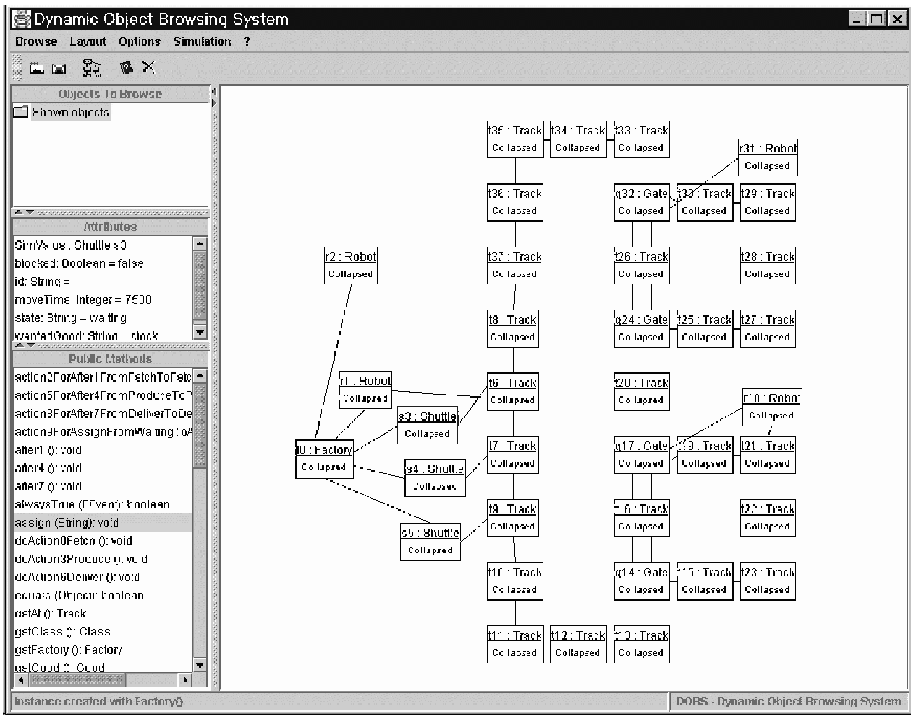

Figure 4 Dobs displaying sample factory as UML object diagram

factory object itself, tracks, shuttles, and assembly lines. For example, tracks are objects of type Track connected with lines representing the links in the object structure[1]. Dobs uses the Java runtime information retrieval techniques, i.e. the reflection API, to extract the internal object-structure of the virtual machine. This observation only allows to get snapshots from the internal object-structure. The user has to layout the objects manually. For example, if a shuttle moves from one track to the next, only the corresponding line will switch from one track object to the next, the shuttle object will keep its position. The currently assigned good clock of shuttle s3 is represented by an attribute of the object shown in the middle list box on the left hand side of the screen.

To get a more realistic simulation, Dobs is able to interpret a look-up table which is the output of the topology diagram. This look-up table consists of various rules, e.g. how objects should be displayed depending on attribute values. The layout is done in a generic

[1] Usually, Java provides only references between objects, but Dobs uses some heuristics to identify a pair of references as a bidirectional link.

way using connectors placed on the graphical representation of objects. If two objects are connected via a link, the link-ends are mapped to connectors (mappings are also specified in the look-up table). The generic layout algorithm tries to put the connectors as close as possible together. Since connectors are directed, and the layouter is able to rotate icons representing objects, each object listed in the look-up table could be positioned. Figure 5

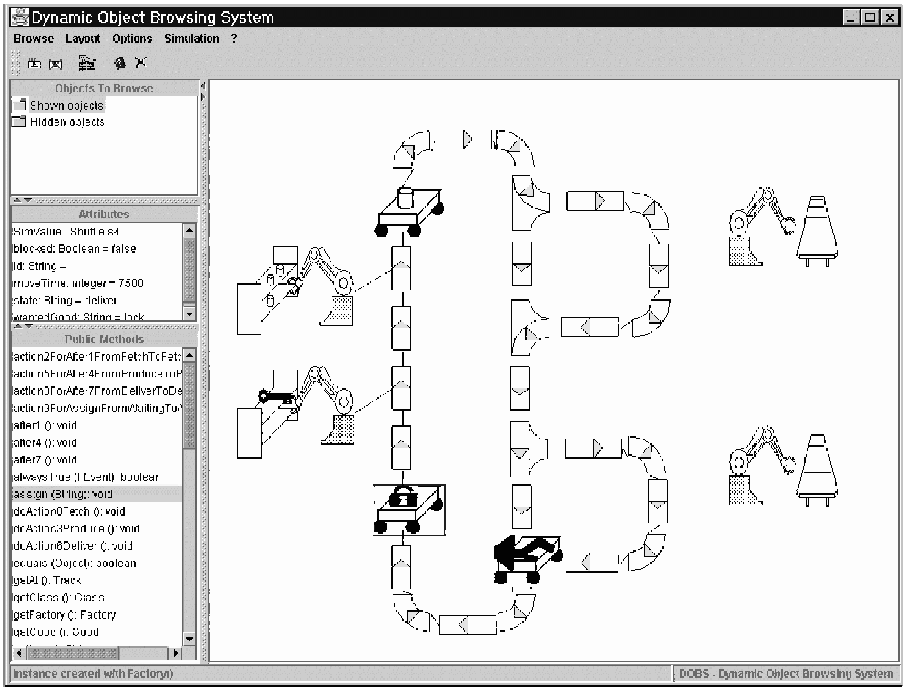

Figure 5 Simulation mode of Dobs with sample factory

shows Dobs in the simulation mode for our sample factory example. In contrast to Figure 4, shuttles are moving on the tracks, assembly lines and storages are easily distinguishable. Likewise, attribute values are visualized, e.g. the upper right transfer gate is currently in fork direction.

4.1 Interaction opportunities

Dobs uses the Java reflection API to display the internal object-structure of the Java virtual machine and allows the user to change object values. This works either for basetype attributes, for example int or boolean, as well as for object reference attributes. Both is done by method invocation of certain objects. As mentioned above, attributes and associations are mapped to Java attributes with appropriate access methods. This allows the user to change the object structure itself. Likewise, Dobs allows to create new objects and to link them to other objects.

For example, Figure 4 and Figure 5 have been produced by first creating a factory object and by calling the build method, which creates the initial object structure. To let the shuttles produce a certain kind of good, a shuttle must receive an assign event. Events are mapped in the source code generation process to appropriate methods in the corresponding classes.

The user can invoke the assign method (lowest list box on the left-hand side) of the selected (highlighted) shuttle s4, cf. Figure 5. Necessary parameters must be specified by the user. On the left-hand side of Figure 6, the assign method is called and the wanted good is passed as a string. On the right-hand side, the shuttle is set on a certain track calling the setAt method and passing the track as parameter.

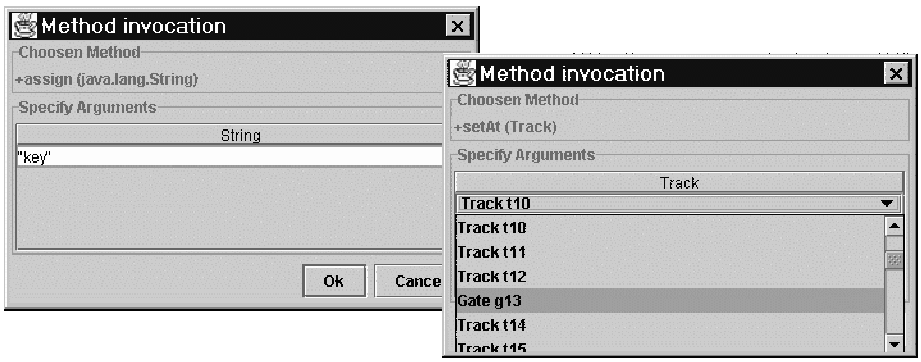

Figure 6 Method invoking with parameters

Overall, Dobs allows the user to test the specification, e.g. to test the robustness of the specification in cases of unforeseen errors. For example, if an assembly line is out of order, the simulation shows directly if the shuttles react appropriately or not. Other opportunities are optimizations, which could be made during the simulation. If the material flow system has a bottleneck, e.g. long queues in front of the assembly lines, the user may enhance or relayout the track system or add additional assembly lines and look if the improvements are effective or not. Especially optimizations and re-configurations of a material flow system force a reconstruction of the real system and take many testing time. This could be analysed during a simulation which reduces the down time of the production.

5 Simulation Kernel

In the previous chapter, we have introduced Dobs as an interactive front-end to visualise the simulated material flow system, which consists of two main parts: the hardware and the software. As already mentioned, we employ Fujaba to specify the control software itself. On a real system, this software controls a node which is in turn responsible for the assigned module. The application software communicates via a process interface with the hardware of the material flow system.

We implemented a simulation kernel to achieve a correct simulation of the physical environment. Thus, the control software can interact with its environment like in the physical production system without any modifications. Note that the kernel encapsulates physical processes like turning a switch or moving a shuttle. In a real system, such operations require some time. Therefore, our simulation kernel has to simulate time aspects. Internally the simulation kernel works in a process-oriented way. It assumes that the processes of a system behave in a cyclic way. The description of such a process in terms of simulation includes all activities which are relevant for it. A scheduler manages the coordination of the processes. A queue keeps the processes and their activation times. The scheduler de-

queues the process with the earliest activation time and activates it. After changing its state and executing all relevant actions, the process determines the next activation time, if necessary. Figure 7 illustrates how the application software controls the simulated system via the interface process of its node.

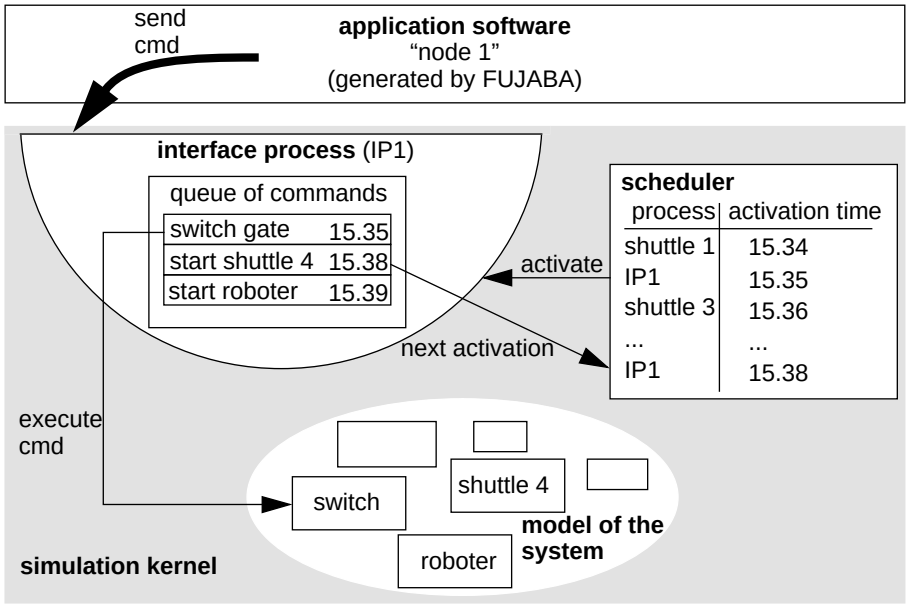

Figure 7 Scheduling processes in the simulation kernel

The application software sends commands to the interface process which simulates the process interface of the real system. The interface process manages a queue with all received commands and their arrival time which depends on the virtual simulation time. In the example the scheduler activates the interface process 1 (IP1) at time 15.35. IP1 now dequeues the next command *switch gate* and executes it. This effects that the gate connected to node 1 is switched. In the next step, IP1 determines the point of time of its next activation and notifies the scheduler which enlists this information in its lookup-table. After that, the interface process deactivates itself and the scheduler activates the next process.

In summary, we have a hardware simulation model of the material flow system and the process interface. This enables us to generate the same code for the control objects, no matter whether we use it in the real production control system or for simulation purposes. For more details see [Sch00].

But, to be honest, the kernel does not have the capability to simulate hardware error conditions of the material flow system, i.e. a malfunction in a switch drive, or a power down of a control node. This is very important if one wants to check the robustness of the control software concerning the reliability of the system.

6 Related Work

The modelling approach underlying this paper is described in our previous work [KNNZ00] in detail. We use UML and SDL to define the static as well as the dynamic behaviour of reactive systems. In general, UML class diagrams specify the static behaviour and for the dynamic behaviour story-charts are used. Story-charts are a combination of statecharts and story diagrams, whereas story-diagrams are introduced in [FNTZ98]. The graph grammar semantics underlying the collaboration diagrams stem from [SWZ95] and [Roz97]. The whole modelling approach is supported by the Fujaba environment and allows to generate executable Java code.

A task-net modelling approach for production control systems is described in [NZ99]. This approach models a production control system with autonomous 'agents', whereas the execution is supervised by a controller to coordinate the independent acting agents. The approach is comparable to [KNNZ00] where statecharts are mapped to an object-oriented task-net like Java implementation and 'agents'(reactive objects) refer to threads.

There is a vast literature on the subject of simulation, e.g. [Zel92] and [Lie95]. Due to the high complexity of production control systems, the analysis of such systems is hard to manage. Thus, simulation systems are used for the analysis purposes [Rei72]. Simulation environments like Simple++ [Tec98] support the design an simulation of a manufacturing process. However, the specification of the control software is not based on an integrated object oriented modelling approach which is formally defined [KNNZ00].

7 Conclusions and Future Work

One of the main problem of todays industry is that the demanded flexibility forces frequent re-configurations of the manufacturing systems. This mainly relies on the fact that the software is tested on the real hardware, which causes long down times. In this paper, we have described an approach to simulate the control software beforehand. To reduce the testing phases on the real hardware, the control software can be validated before its use in our simulation environment. The environment observes the running code by using the Java reflection library. Thereby, the production sequences can be visualized and analysed. The simulation is based on a simulation kernel, which serves as a model for the hardware of the production system.

Our simulation kernel can be described as a discrete process interaction simulation with variable time increments. Discrete simulation approaches are often used in business economics for example. Additionally, most software packages for simulating production systems just support discrete approaches. Although the event scheduling approach can be implemented much easier, the process interaction approach is more suitable for the modular character of the discussed production system.

As a result, a prototype implementation of the simulation environment has revealed some mistakes in the control software specification made in the ISILEIT project for the sample factory.

Although we are able to visualise the running system in 2D, the engineers of our department aspire to display the whole system in 3D. Moreover, we will add the possibility to display some statistics concerning the performance of the system. Furthermore, the possi-

bility of checking the event-flow between the distributed, asynchronously communicating objects should be added for debugging facilities.

Finally, the simulation lacks a model of the communication bus and the bus interface. This becomes very important if intense traffic on the communication bus leads to communication delays or errors.

Acknowledgements

Many thanks to Matthias Gehrke, Ralf Schomaker, Jörg P. Wadsack, Albert Zündorf for their fruitful discussions, careful proof readings, and a lot of suggestions to this work. Special thanks go to Martin Glinz and all reviewers of this paper, who helped to improve it with their comments and suggestions.

References

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In G. Engels and G.Rozenberg, editors, *Proc. of the 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*. Springer Verlag, 1998.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pneuli, M. Politi, R. Sherman, A. Shtull-Tauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. In *IEEE Transactions on Software Engineering*, pages 403–414. IEEE Computer Society Press, 1990.

[KNNZ00] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the 22th Int. Conf. on Software Engineering (ICSE), Limerick, Irland*. ACM Press, 2000.

[Lie95] F. Liebl. *Simulation. A problem oriented introduction (in german)*. Oldenbourg, Munich, 2nd edition, 1995.

[NZ99] J. Niere and A. Zündorf. Using Fujaba for the Development of Production Control Systems. In *Proc. of Int. Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands*, LNCS. Springer Verlag, 1999.

[Rei72] A. Reinhardt. *Simulation eines Fertigungsprozesses in GASP*. Großmann, 1972.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, Singapore, 1997.

[Sch00] R. Schomaker. *Development of a simulator for decentralised controlled, track-bounded transport systems (in german)*. Master's thesis, University of Paderborn, Department of Mathematics and Coputer Science, Paderborn, Germany, 2000.

[SWZ95] A. Schürr, A.J. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. In W. Schäfer, editor, *Software Engineering - ESEC '95*. Springer Verlag, 1995.

[Tec98] Technomatix. *Technomatix: Reference Manual, SIMPLE++ 5.0, Handbuch zur Simulationssoftware*, 1998.

[Zel92] M. Zell, editor. *Simulation based manufacturing control*. Oldenbourg, Munich, 1992.