Towards an Integrated System Model for Testing and Verification of Automation Machines*

Benjamin Hummel
hummelb@in.tum.de

Peter Braun
braunpe@in.tum.de

Institut für Informatik, Technische Universität München, Garching b. München, Germany

ABSTRACT

Models and documents created during the development of automation machines typically can be categorized into mechanics, electronics, and software. The functionality of an automation machine is, however, realized by the interaction of all three of these domains. So no single model covering only one development category will be able to describe the behavior of the machine thoroughly. For early planning of machine design, virtual prototypes, and especially for the formal verification of requirements an integrated functional model of the machine is required. This paper introduces a technique which can be used to model automation machines on an abstract level, including coarse-grained descriptions of mechanics, electronics and software aspects with special focus on modeling domain-specific issues such as material flow and collision response. The resulting models are detailed enough to be simulated or verified but still suitably abstract to allow fast creation and efficient simulation.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Specifications—*Languages*; J.7 [**Computers in other Systems**]: Industrial control

General Terms

DESIGN, LANGUAGES

Keywords

Modeling, Automation Engineering

1. INTRODUCTION

An important step during the development of any system is to check whether the system built actually adheres to its requirements. This check can either be performed by verification techniques, which are basically (machine supported) proofs of certain properties of an abstract model of a system, or by testing, which checks the properties only for a finite set of inputs, and consequently can only show errors of a system but never its correctness.

When looking at automation machines, *e. g.,* for brick production, a system consists not only of software, but also of mechanical parts, sensors, and actors (to which we refer as hardware here). Thus it is usually not sufficient to test the software and hardware in isolation, but especially their interaction has to be taken into consideration. For example an object passing a sensor might trigger some software routine, whose decisions cause some actor to influence the mentioned object in some way.

Traditionally such machines could only be tested after assembly when both the hardware and software have been completed. This causes several problems, as any defects found after assembly usually require more effort to be fixed, potentially dangerous or destructive tests even might not be performed at all due to the high cost of the machine, and the testing process is hard to automate or parallelize. During the last decade several approaches have been proposed, which include running real software (*i. e.,* final code on the actual controller) using a simulated machine (also known as virtual commissioning), or simulating both software and hardware[1] [1, 14]. However all of those approaches usually treat the software and hardware separately, which is to some part imposed by the development processes and tools in the area of automation engineering which still separate mechanical, electrical, and software design very strictly.

1.1 Problem Statement

We strongly believe that for understanding and reasoning about automation machines, a single integrated model covering all of the mechanical, electrical, and software aspects is needed, as only the connection of hardware and software behavior describes the functionality of the machine adequately. This is in consensus with Jackson [10], who argues that requirements usually capture not the behavior of the software itself but the effect of the software to its environment. In this paper we are trying to shift the boundary of our models to include more of the software's environment, which happens to be the machine's hardware.

We especially concentrate on the early planning phase of automation machines, where we have to deal with coarse-grained models of mechanical, electrical and software as-

*Parts of this work are the result of a research cooperation with Siemens CT and Siemens A&D.

[1]The combination of real hardware using simulated software seems not to be actually used. A reason might be that such a setup does not make potentially destructive tests more safe.

pects, as we think that early integration of these aspects will help to get a single-minded development process. Furthermore, such an abstract model could be used for testing (which includes simulation), generating test-cases, or even performing formal verification. As visualization and simulation of the machine including material flow and collisions is of central concern, dynamical and geometrical aspects should be captured by the abstract model.

To the best of our knowledge no modeling technique supporting the integrated modeling of software and hardware in a way suitable to formally analyze automation machines exists. Thus our goal is the development of such a technique. Obviously this would not be an entirely new language, but rather a combination of several established formalisms, which is then forged to the domain of automation machines. Typically most of the information captured by the integrated model is already available distributed over various models and documents. Our model is used to gather those parts describing the machine's behavior and its interaction. This paper documents the current status of our proposal of such a modeling language.

1.2 Outline

The next section introduces the domain of automation machines (more specifically production machines), followed by a list of requirements resulting from the the domain's specifics. Section 3 introduces our system model, succeeded by some examples in Section 4. Finally we discuss the relationship of related work to our approach (Sec. 5) and conclude by future work (Sec. 6). For further details and figures we refer to the extended version of this paper [2].

2. DOMAIN ANALYSIS

When proposing a new modeling technique one should always justify, why none of the existing techniques could be used and yet another language had to be designed. To do this, we will first give an overview of the domain of automation machines we are treating here, followed by specific requirements for a modeling technique arising from it. A discussion of related modeling techniques as well as the differences to our approach will be postponed to Section 5.

2.1 Production Machines

While the overall scope of our work are automation machines, we focus on production machinery here, as all machines examined during the development of the modeling technique were from this class. Production machines perform multiple transformation steps on a physical product, thus the focus is on transportation and grouping of material, dealing with congestion, and avoiding collisions between multiple parts of the machine. While other sub-domains, such as machine tools or process technology, are not treated here, we expect to only need minor adjustments to our technique to support them.

The main aspect of production machines is a strong focus on material and material flow, *i. e.,* movement and modification of material (products). Another characteristic is the commonness of collisions between objects. While some collisions are not desired, *e. g.,* between robotic arms, many functions of the machine depend on them. Examples are delaying moving material on a conveyor with a stopper, queuing up material, or starting actions as material passes a photoelectric barrier (collision between object and light ray).

2.2 Model's Requirements

Based on the specifics of production machines and the applications we have in mind, we assembled an informal list of requirements for a suitable modeling technique. This section includes only requirements, which are specific for the domain of automation machines. Other requirements which are quite common for modeling techniques describing reactive systems are not listed. We are not yet sure whether continuous time or a fully hybrid model, which allows components to exchange time continuous functions, are required to formulate useful abstractions of the machine, thus we excluded these from our list of requirements, until further case studies provide clarity.

Explicit Modeling of Material. Material should be supported as a first class entity with position and state to ease the visualization and tracing of the material flow. Especially the exchange of material between components should not be "simulated" using just status signals, as this makes the model hard to comprehend for domain experts.

Implicit Collision Detection, Explicit Collision Handling. Detecting collisions of arbitrarily shaped and oriented objects in three dimensional space is not an easy problem. As such it should not be the responsibility of the user to explicitly model collision detection, but rather the execution environment of the model should deal with it. Contrary the reaction to a collision, especially whether it is allowed or understood as harmful, has to be modeled explicitly, as often only the domain experts can decide whether a collision is expected. Furthermore this makes collisions implicitly realizing parts of the functionality more visible.

Geometric Data. To make the detection of collisions possible, the components and material should have geometric information attached. As the model is an abstract view of the machine, coarse geometry should be sufficient for most purposes. This geometric information can help visualizing the model, which is a key requirement for making the model understandable to non-experts.

3. THE SYSTEM MODEL

This section introduces the modeling elements used and the intuitive meaning of them as far as possible. The system model has been heavily influenced by FOCUS [4] and its tool implementation AutoFOCUS [3]. Some examples are provided in Section 4.

We include explicit error states in our model which we call *error conditions.* One goal of testing or verification is to show that these conditions never occur. Two typical situations for errors are inconsistent material flow or collisions that were not anticipated and specified by the modeler.

3.1 High Level View of the Model

The main elements of our model are quite similar to those of other component-based descriptions, but the difference to other modeling techniques will become obvious in the following sections. A machine is a component, whose syntactic interface is described using ports, *i. e.,* points on which information (in the broadest sense) is exchanged with the environment. This exchange is made explicit using channels, linking an input to an output port. Ports and channels have a type, defining the kind of information dealt with. Only ports of compatible types may be connected.

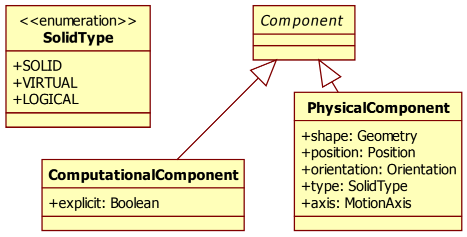


Figure 1: The different component types

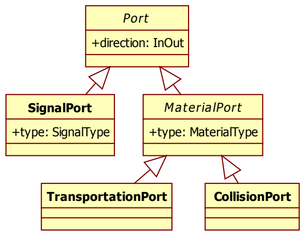


Figure 2: The different port types

A component can be specified either by composition or by primitive specification. Composition means, that the component is described by other components and their interplay, which leads to a hierarchical decomposition of the entire machine. The specified component then acts as a wrapper around its inner components, which in turn connect to the ports of the surrounding component. If a component is not split into further subcomponents, it has to be specified directly. We suggest the usage of hybrid automata, although other techniques could be used in conjunction (*e. g.,* table based descriptions).

3.2 Component Types

For making the purpose of a component more explicit, we distinguish between computational and physical components as shown in Figure 1. *Computational components* are used for describing purely logical aspects of a machine, such as computations and decisions implemented in code or controllers (which might be part of the hardware in the real machine). Subcomponents of a computational component also have to be computational.

Physical components represent those parts of the machine residing in the physical world, *i. e.,* having position, orientation, and shape. They may contain further physical and computational components, which allows us to keep parts of the machine and the corresponding controller code together.

For the physical components there are three sub-types. One are *solid* components, which means that the space described by the component can not be penetrated by other solid objects. These are used for modeling actuators or structural mechanics. Next are *virtual* physical components, which are not solid geometry but rather penetrable regions. They are used for modeling light beams of sensors, or ranges of RFID scanners. Finally there are *logical* components, used for grouping other physical components.

3.3 Interface Description

As already stated, the syntactic interface of components is specified using *ports*. Ports are labeled with a type and can be connected using *channels*. Only ports of compatible types may be connected. To enforce encapsulation, only ports of components within the same scope (*i. e.,* either being all top-level components or being subcomponents of the same component) may be connected. The ports of a component are available as inner ports to its subcomponents with changed orientation, *i. e.,* input ports becoming output ports and vice versa, and act as a repeater for the signals or material exchanged, *i. e.,* have syntactical meaning only.

Similar to the components we differentiate between several kinds of ports (Figure 2). *Signal ports* are used for exchanging logical signals, modeling asynchronous function calls or bus communication. For signal ports each output port may be connected to any number of input ports, as data can be easily duplicated. However each input port may only have one source output port, making merging of signals explicit. The type of a signal port is described using classical type systems (*e. g.,* algebraic specifications [5]).

The remaining ports deal with material and may only be used for physical components. As a type for those ports the kind of material exchanged is given, where material is currently only specified by its shape and some visualization properties (such as color). On the long term however extensions for material state are planned. The material ports are further divided into *transportation ports* and *collision ports.* Transportation ports describe the actual flow of material, *i. e.,* material is passed between components along these ports. As material can not be duplicated, an output transportation port may only be connected to one input port. However (dual to the signal ports) the input ports may be connected to any number of output ports, as material has a position and thus merging of multiple objects is straightforward. Collision ports are used for describing which collisions are allowed (Section 3.7). There are no restrictions on the number of connections for collision ports.

3.4 Functional Description using Communicating Hybrid Automata

To describe the behavioral aspects of a primitive component, hybrid automata which use the signal ports of the component for communication are proposed. The automaton model used is influenced mostly by the one used in AutoFocus [3] and extended using ideas from [8]. An overview diagram and details are given in [2].

The main difference over the automata used in AutoFocus is that each control state has a list of flow conditions, which are first order differential equations defining the derivatives for variables of continuous (double) type which are applied while the automaton is in this control state. Additionally there are some extensions described in the following sections making them also define behavior for material and material ports.

3.5 Geometry and Motion

To support visualization of the machine and detection of collisions, geometric information is incorporated into the model. A possible model which is just composed of geometric primitives is shown in Figure 3, however more complex geometry models could be used as well, provided there are algorithms for testing them for collisions.

All physical components have position, orientation, and geometry. Additionally components can be augmented by at most one motion axis describing one degree of freedom

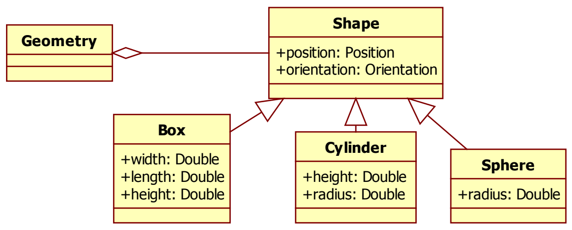


Figure 3: Simple model for describing geometry

(linear or rotational). The axis has a direction of motion (or rotation) and associated minimal and maximal values. The current position along this axis is made available to the hybrid automaton of the component, which can then assign new values to it or set its derivative in the flow conditions.

For modeling more complicated movements, a motion hierarchy is used. The motion hierarchy describes for each physical component its motion predecessor (if one exists). A component always follows the movements of its motion predecessor. This way joints with more than one degree of freedom can be modeled. The motion hierarchy is modeled explicitly for all physical components within one component by giving for each one its motion predecessor. For components without an explicit motion predecessor, their owning component is the motion predecessor.

As described before, the motion of a component affects both the component itself and its children in the motion hierarchy. For some components (*e. g.,* conveyor belts) we want the movement to only affect children (and material as described in the next section) but not the component itself. We call such components *static*. Static components can still be moved by their motion predecessors.

3.6 Dealing with Material

So far we described how components are modeled, but mentioned material only in the context of material types for ports. This is because the instances of material (which we call *material objects*) are specified indirectly by the model and its behavior and are only used during the execution of the model. This is similar to "normal" types, where we annotate ports with types while the actual values (instances of a type) are seen during simulation only. As material is central to our models and (contrary to typed signals) its usage has not been described in the literature before, we will detail its usage and handling in this section.

Material objects are instances of a material type and are physical objects, that is they have position and orientation. The geometry is given by the material type. Each material object is managed by exactly one physical component, which we call its *owner* and which controls the life cycle of material objects controlled by it. There are *new* and *delete* operations, which are used for creating material or deleting owned material in a transition of the component's automaton. The *new* operation takes the material type created and initial position and orientation relative to the position of the creating component. The delete operation gets the set of components to delete as a parameter. This set is defined using predicates evaluated on all owned material objects. Material can be selected based on type, state, or location. To simplify the definition of location, a component defines *locators*, which are geometric objects positioned relative to the component (*i. e.,* following every movement). The predicates may select material objects intersecting a given locator.

The motion of material objects is defined by their owning component. For the physical motion they follow every move of their owner component, unless their path is blocked. The logical motion (*i. e.,* the change of ownership) is supported by the operation *push*, which takes a set of owned material objects and a transportation output port. All objects are then transfered to the target component, which becomes the new owner. The target component is unique, as transportation output ports are connected to at most one input port. If no input port is present this is an error condition. A component may reject new material by means of *acceptance filters*, describing which kinds of material objects are accepted from which transportation port. These filters can be set for each state, so for example a gripper might accept material while it is near the ground but not while in midair. If material is pushed to a component which does not accept it, this also is an error condition.

3.7 Collisions and Collision Response

As stated in Section 2, collisions should be explicitly treated by our model. The detection of collisions is the task of the execution environment (simulator or verifier), while the response to such collisions has to be modeled explicitly. There are three different cases of collisions. First are collisions between two material objects, which cause a material object to be stopped if its path of motion is blocked. Additionally the material objects' owners are queried whether material managed by them may collide. If any of them disallows collisions (which may depend on the current state), this is an error condition. This way one can model parts of the machine where collisions are allowed (*e. g.,* queuing up material) or disallowed as they might lead to damage or congestion.

Second are collisions between material object and physical component. If there is a channel between the material object's owner and the colliding component, a reference to the colliding object is made available on the corresponding ports, which then can be queried in the transitions of the automata of the colliding component and the owning component. If no such channel exists, this is an error condition, *i. e.,* such a collision is not allowed. Additionally the material object is stopped, if a solid component blocks its way.

Finally for collisions between physical components the response depends on the solidity of the components. If both components are solid, this is an error condition. If none of the components is solid, the collision is ignored. For collisions between solid and non-solid components, the situation is similar to the second case. The collision does not lead to an error condition only if there is a collision channel from the solid to the non-solid component. The type of this channel has to be *component object*. The collision is then reported on the corresponding collision port.

4. EXAMPLES

This section contains examples how various parts of an automation machine can be modeled using our approach. To provide these models, a concrete syntax is required. For the automata we propose the usage of the usual ellipses connected by arrows, for components we use boxes (labeled with the type of component) and show ports by symbols on the border of the components. The symbols we are using in here are summarized in Figure 4. All other elements, such

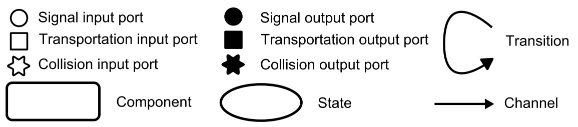


Figure 4: Symbols used for components and ports

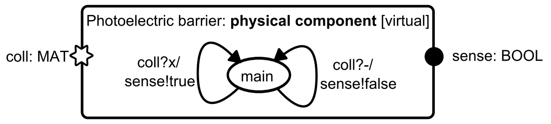


Figure 5: Model of a photoelectric barrier

as transition conditions, are given textually. We will not provide a concrete syntax for them, but hope that they are understandable from the examples.

4.1 Photoelectric Barrier

A photoelectric barrier consists of a source of directed light and a receiver unit. If something is blocking the path between source and receiver, the receiver detects this and sends corresponding signals. Thus this barrier is just a converter from collisions (with the light beam) to signals.

A possible model is given in Figure 5. The collision input port is of some type MAT and the signals sent are boolean values. The geometry attached to the component (which is not shown) resembles the area of the light beam. The light source and the receiver are not relevant to us, and thus not modeled. The automaton consists of a single state only. If some material collides (is "received" at the collision input port (`coll?x`)) we send *true* to indicate detected material, otherwise (we use `coll?-` to check for no values at the port) the value *false* is sent on the `sense` port.

4.2 Conveyor Belt

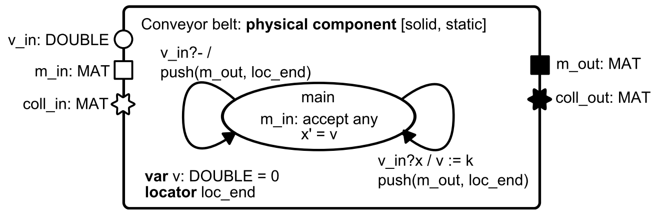
Our next example is a conveyor belt for material MAT, shown in Figures 6 and 7. It has one signal input accepting a value for transportation speed. In the single state of the automaton the derivative of the motion of the component `x'` (its speed) is set to `v` and it is defined that all material entering the conveyor at `m_in` is accepted. As we only want the material to move and not the conveyor belt itself, the component is marked static. The direction of motion is given as a vector (linear axis) in the geometric description.

The first transition of the automaton pushes all material intersecting with the locator `loc_end` to the next component connected to `m_out`. This locator is also defined in the geometric view and is at the end of the conveyor. The second transition is active if a new speed value is present at the `v_in` port, which is stored in `v` and material is pushed forward just as in the first transition.

4.3 Grouping Unit

Finally a grouping unit is composed from the components defined so far using a logical component. The purpose of this unit is to preprocess a stream of material objects with random distances between them and form a material stream consisting of groups of n material objects with each group separated by a gap of given length.

One solution to this problem is to use two conveyor belts positioned next to each other, whose speed is carefully con-



Conveyor belt: physical component [solid, static]
v_in: DOUBLE
m_in: MAT
coll_in: MAT
v_in?- / push(m_out, loc_end)
main
m_in: accept any
x' = v
v_in?x / v := k push(m_out, loc_end)
var v: DOUBLE = 0
locator loc_end
m_out: MAT
coll_out: MAT

Figure 6: Model of a conveyor belt

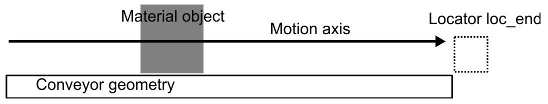


Figure 7: Simplified geometry of the conveyor belt

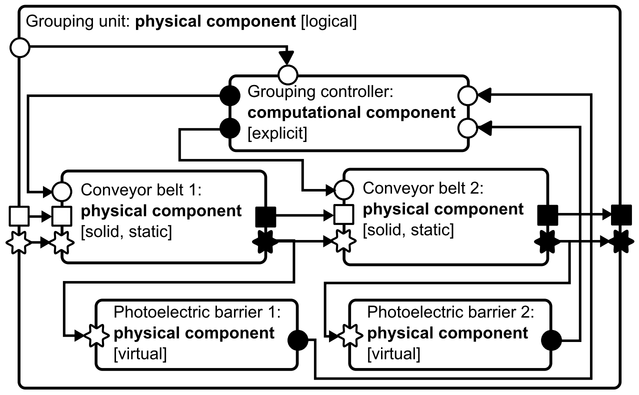
trolled to form the required groups. Basically we keep material at the beginning of the second conveyor and halt it, until the front most material object on the first conveyor has the correct position. Then both conveyor belts move at same speed until one more material object is handed over. Position of material objects on both conveyors is located by photoelectric barriers. A realization is shown in Fig. 8, where conveyor belts and photoelectric barriers are the components described before. The controller is a computational component adjusting the speed of the conveyors based on signals from the photoelectric barriers and commands received via a third input port (specification omitted).

5. RELATED WORK

Our work was heavily influenced by the FOCUS modeling theory [4] whose primary purpose is the description of reactive systems. There also is a modeling language and a tool based on FOCUS, called AutoFOCUS [3]. Our entire high-level meta-model is similar to the one of AutoFOCUS and if we eliminate physical components and material ports from our model the remainder is nearly the same.

For describing the software part of the machines we also could have used any of the well-known approaches for modeling systems, such as state charts [7], I/O automata [11], Petri nets [13], process algebras (*e. g.,* CSP [9]), or any of the many variants of them including hybrid extensions.

However none of the techniques listed so far has a notion of material, geometry, or collision detection and response.



Grouping unit: physical component [logical]
Grouping controller: computational component [explicit]
Conveyor belt 1: physical component [solid, static]
Conveyor belt 2: physical component [solid, static]
Photoelectric barrier 1: physical component [virtual]
Photoelectric barrier 2: physical component [virtual]

Figure 8: Model of a grouping unit

Thus they can in principle be used to describe these machines, but place a high burden on the modeler who has to close the gap between these domain concepts and the provided (rather low level) modeling concepts.

SysML (Systems Modeling Language) [12] is a language for modeling systems by the Object Management Group. It is based on UML and as such shares some of its drawbacks, such as (partially) unclear semantics and an overwhelming multitude of modeling constructs. Additionally the objections given for all other techniques before also apply here, as support for material flow and collisions is missing.

Simulation models used for virtual commissioning (software-in-the-loop test with controller hardware and simulated machine) are another source for modeling techniques. Existing industrial solutions however do not couple the geometry tightly to the machine's components, but only describe reactions of a visualization model to state changes of the machine. For example the SEMI project (Simultaneous Engineering for Development of Machines with Micro-systems) [1] used ROOM and C++ for modeling the simulated machine. Consequently support for continuous changes within the machine, material flow, or collision detection and response had to be simulated somehow by the modeler.

Collision detection and response are one of the main areas treated by physical simulations. A common drawback is their missing support for discrete changes, which may be caused by collisions or logical signals. The modeling language Modelica [15] supports both continuous flows and discrete changes (hybrid modeling). For collision detection and response an extension is proposed in [6]. What makes these models inappropriate for our purposes is the high level of detail required for physically exact simulation (*e. g.,* mass, moment of inertia, friction coefficients) which is often not available during early planning stages. Furthermore this makes it hard to model parts of the machine at different levels of abstraction, as we always have to respect physics (which we may violate in our models).

6. CONCLUSION AND FUTURE WORK

In this paper we argued in favor of an integrated system model for describing automation machines in a manner suitable for simulation and reasoning about the machine (testing and verification). We presented requirements for such a modeling language and proposed a possible technique for describing such machine models. Of course the model presented in this paper is only the first step towards a methodology for testing and verifying automation machines, and as such still leaves many open questions.

A first set of questions deals with extending or refining our system model, *e. g.,* by including continuous channels between components (*i. e.,* instead of single values, functions over time are transmitted). Before taking further steps in this direction however, we first have to gain more experience in modeling different kinds of real machines to better understand the limitations of our approach. This is also a precondition for extending the language from production machines to other sectors of automation machines, such as machine tools or process technology, for which we especially expect to extend the material type system, *e. g.,* supporting non-integral material quantities.

The other set of questions covers actual usage of the model, for which we might need the definition of formal semantics. These include test-case generation and verification of properties, which were the driving applications for creating our model. However, there are other possible applications of such a model, starting from discussing requirements with a customer or acting as the main point of communication during the development process, to using the model for the analysis of possible failures or planning of maintenance tasks. Some of these applications might require extensions to the model, such as annotations of fault behavior and error probability for components.

7. REFERENCES

[1] J. Albert, K. Bender, T. Holzmüller, B. Jünger, O. Kaiser, W. Kriesel, O. Prinz, C. Schaich, J. Schullerer, and J. Tomaszunas. *Echtzeitsimulation zum Test von Maschinensteuerungen.* Herbert Utz Verlag, 1999.

[2] P. Braun and B. Hummel. Towards an integrated system model for testing and verification of automation machines. Technical Report TUM-I0802, Technische Universität München, 2008.

[3] M. Broy, F. Huber, and B. Schätz. AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, 13(13):121–134, 1999.

[4] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer, 2001.

[5] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *Monographs in Theoretical Computer Science.* Springer, 1985.

[6] V. Engelson. *Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing.* PhD thesis, Linköpings Universitet, 2000.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[8] T. Henzinger. The theory of hybrid automata. In *Verification of Digital and Hybrid Systems*, NATO ASI Series F: Computer and Systems Sciences 170, pages 265–292. Springer, 2000.

[9] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[10] M. Jackson. Testing the machine in the world. In *Proc. of Haifa Verification Conference 2006*, pages 198–203, 2006.

[11] N. A. Lynch. Input/output automata: Basic, timed, hybrid, probabilistic, dynamic, .. In *Proc. 14th Int. Conf. on Concurrency Theory (CONCUR'03)*, pages 187–188, 2003.

[12] Object Management Group. OMG SysML specification v. 1.0, May 2006.

[13] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Universität Bonn, 1962.

[14] W. Schloegl. Bringing the digital factory into reality – virtual manufacturing with real automation data. In *Proc. of International Conference on Changeable, Agile, Reconfigurable and Virtual Production*, pages 187–192, 2005.

[15] M. Tiller. *Introduction to Physical Modeling with Modelica.* Springer, 2001.