TUM

INSTITUT FÜR INFORMATIK

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

Peter Braun and Benjamin Hummel

TECHNISCHE UNIVERSITÄT MÜNCHEN

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

Peter Braun and Benjamin Hummel

Institut für Informatik,
Technische Universität München
{braunpe,hummelb}@in.tum.de

Abstract. The models and documents created during the development of automation machines typically can be categorized into mechanics, electronics, and software/controller. The functionality of an automation machine is, however, usually 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 the 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 modeling technique which can be used to describe automation machines on an abstract level, including coarse-grained models of mechanics, electronics and software aspects. The resulting models are detailed enough to be simulated or verified but still abstract enough to allow fast creation and efficient simulation.

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 (usually machine supported) proofs of certain properties of the system (or an abstract model of it), 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[1].

When looking at automation machines, 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 again 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,

---

⋆ Parts of this work are the result of a research cooperation with Siemens Corporate Technology and Siemens Automation and Drives.

[1] "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." (E. W. Dijkstra in [Dij72])

as any defects found after assembly usually require more effort to be fixed, potentially dangerous or destructive tests 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[2] [ABH+99]. 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. We especially concentrate on the earlier planning phase of automation machines, where we have to deal with coarse-grained models of mechanical, electrical and software aspects, as we think that early integration of the 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. Core issues of this abstract model is the coarse-grained visualization and simulation of the automation machine including the material flow and the possibility to deal with collisions. Therefore dynamical and geometrical aspects should be captured by the abstract model.

As no modeling technique supporting the integrated modeling of software and hardware in a way suitable to formally analyze automation machines is known to us, 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 used for describing a machine. However, our model is used to gather those parts describing the machine's behavior and especially to describe their interaction. This paper documents our proposal of such a modeling language.

1.2 Outline

In the next section we will give a short overview of the domain of automation machines and more specifically production machines, followed by a list of re-

---

[2] It seems that the last possible combination of real hardware using simulated software is not actually used, as it would require real-time systems which are already provided by the real controller. Additionally such a setup still does not make potentially destructive tests more safe.

quirements resulting from the specifics of the domain and the applications we have in mind. Section 3 introduces our system model, succeeded by an example model in Section 4. Afterwards we present related work and discuss the similarities and differences to our approach (Section 5) and conclude by expected future work (Section 6) and a short summary (Section 7).

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 the modeling language are automation machines, we will focus on production machinery here, as all machines examined during the development of the technique were from this class. Production machines usually execute multiple transformation steps on a physical product, thus the focus is on transportation and grouping of material, dealing with congestion, and avoiding collisions between the multiple parts of the machine. While other sub-domains, such as machine tools and process technology, are not treated explicitly, we expect to only need minor adjustments to the modeling technique to support them. We discuss those extensions in Section 6 as far as we are aware of them.

The most prominent aspect of production machines is the strong focus on material and material flow, as the movement and modification of this material (products) is their main purpose. Another characteristic is the commonness of collision between objects in the physical world. While some collisions are obviously not desired, such as those between two robotic arms, many functions of the machine are explicitly realized by exploiting collisions. These include delaying of moving material on a conveyor belt by using a mechanical stopper, queuing up material, or starting other actions as soon as material passes a photoelectric barrier (which boils down to a collision between the object and a 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, which is given next.

1. **Hierarchic components**
   We want to decompose the system into components which can be reused later. This not only simplifies the understandability of the model, but also

allows dividing the modeling task to more than one person. By creating new components as a composition of existing ones, the machine is described as a hierarchy of components.

2. **Clearly defined interfaces**

   Again to ease reuse and understandability, the interfaces of the components should be clearly defined. We do not want components to interact in some way only because they happen to be physically placed right next to each other without any explicit notion in our model. This becomes especially tricky with the physical parts of the machine, which might influence other parts by colliding with them. Such interfaces will also aid in verification, as we can check properties on the components and be sure that the properties are still valid after composition.

3. **Explicit modeling of material**

   Material should be supported as a first class entity. It should have a position and a 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 the domain experts.

4. **Implicit detection of collisions, but 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 (simulator) of the model should be responsible. Contrary the reaction to a collision, especially whether it is allowed or understood as harmful, has to be modeled explicitly in the model, as often only the domain experts can decide whether a collision is expected. Furthermore this makes those collisions, which implicitly realize parts of the functionality, more visible.

5. **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. Additionally this geometric information can help in a visualization of the model, which is a key requirement for any model simulator to make the model understandable to non-experts.

6. **Suitable level of abstraction**

   We want to be able to model at least parts of a machine on a high level of abstraction. On the one hand this allows faster creation of the models, as we do not have to describe details not relevant to us. On the other hand the resources needed for simulation and verification usually increase dramatically with the size of the model, thus abstraction can help to make simulation run fast (*e. g.,* for real-time tests) or verification to be feasible in reasonable time.

7. **Defined semantics**

   For both simulation and verification, the meaning of the model must be clearly defined. The best way for doing this without ambiguity is to give a semantics for the syntactical representation of the model.

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 parts from our list of requirements, until further case studies provide clarity.

As we found no suitable modeling language matching all of these requirements, we decided to design a new one, which is presented in the following section. The drawbacks of other approaches are summarized in Section 5, while we discuss to what extent our proposal fulfills those requirements in Section 7.

3   The System Model

In this section we introduce our system model, *i. e.,* we define the modeling elements used (abstract syntax) and try to give an intuitive meaning for them as far as possible. The system model and its (informal) semantics have been heavily influenced by FOCUS [BS01] and especially its tool implementation AutoFOCUS [BHS99]. As this section is quite abstract, we provide some examples in Section 4.

As the dynamic aspects of an automation machine can result in invalid states, some kind of error handling is necessary. We call such errors which are detected during simulation or analyzing dynamic aspects of the automation machine *error conditions*. As they represent undesirable behavior, one goal of testing or verification is to show that these conditions are never met. A simulation should stop if such an error is encountered, as the further behavior from such a erroneous state is undefined. Two typical situations for errors are inconsistent material flow or collisions which were not anticipated and specified by the modeler, which are described later on.

3.1   High level view of the model

Before going into detail, we give a high-level overview of our system model here. The main elements are shown in Figure 1, which are actually quite generic. The difference to other modeling techniques will become more obvious in the following sections.

A machine in our model is a component, whose syntactic interface is described using ports which are the points on which information (in the broadest sense) is exchanged with the environment. This exchange of information is made explicit using channels, linking an input to an output port. Each port and channel has a certain type, defining the kind of information dealt with. As expected only ports and channels having compatible types may be connected.

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 into components. 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 can not be split into further subcomponents (a primitive

component), it has to be specified directly. For this we suggest and later describe the usage of hybrid automata, although other techniques can be used in conjunction (*e. g.,* table based descriptions).
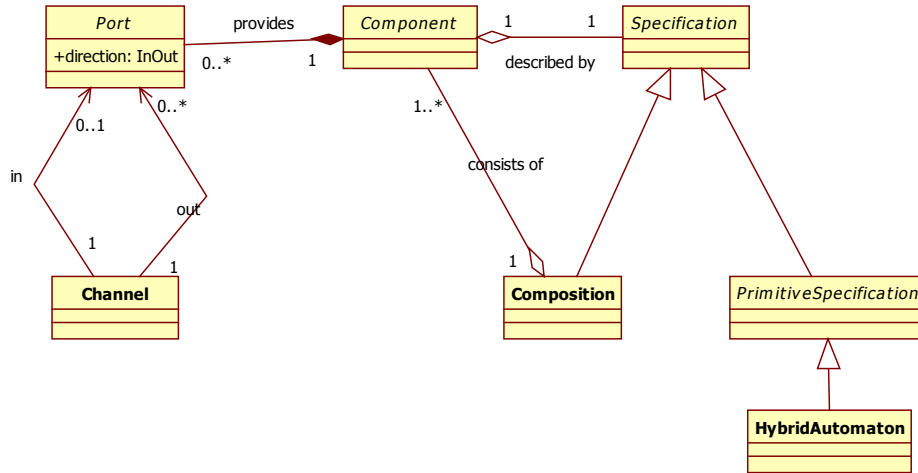

Fig. 1. High-level view of the modeling elements

3.2 Component types

For making the purpose of a component more explicit, we distinguish between computational and physical components as shown in Figure 2. *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). The subcomponents of a computational component also have to be computational components. We annotate computational components with a flag indicating whether they are explicitly implemented in the machine or are part of the implicit behavior of the hardware. While this flag has no semantical meaning, it can be helpful in deciding which parts should be used for automatic generation of source code.

Physical components represent those parts of the machine that are physically present, *i. e.,* have a position, orientation, and a shape, which is generally a compact subset of the three dimensional Euclidean space. 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
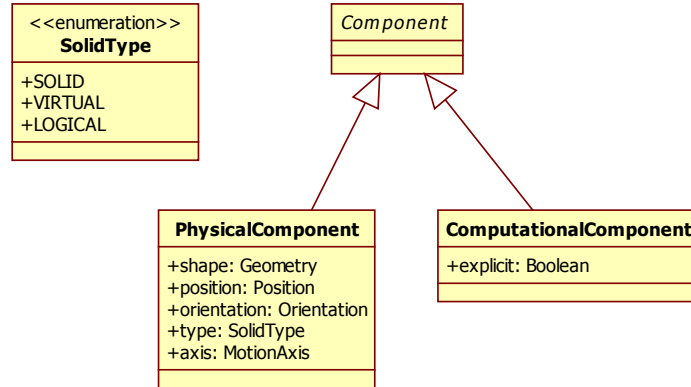
Fig. 2. The different component types supported

geometry but rather penetrable regions. They are usually used for modeling the light beams of sensors, or the range of RFID scanners. Finally there are *logical* components, which are used for grouping other physical components to form a logical unit.

3.3 Interface description

As already stated, the syntactic interface of components is specified using *ports* which describe directed communication points to other components. Ports are labeled with a type describing the kind of object sent over it and can be connected using *channels*. Only ports of compatible types may be connected. To enforce encapsulation, only the 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. However all ports of a component are available as inner ports to its subcomponents (but with changed orientation, *i. e.,* input ports becoming output ports and vice versa) to be connected to them. Thus the ports of a component containing subcomponents act as a repeater for the signals or material exchanged, *i. e.,* have syntactical meaning only. While this might seem to just complicate the wiring of the components, as connections passing component boundaries have to be made explicit using additional ports, this helps in hiding the internals of components and simplifies the replacement of a component, as the entire syntactical interface of the component is known without having to inspect its implementation (in terms of subcomponents).

Similarly to the components we also differentiate between several kinds of ports (Figure 3). *Signal ports* are used for exchanging logical signals. They are used for modeling asynchronous function calls or communication over some kind of bus. For signal ports each output port may be connected to any number of input ports, as the data can be easily duplicated. However each input port may only have one output port as source, as there are many possibilities for merging these signals and thus this merging should be made explicit inside of
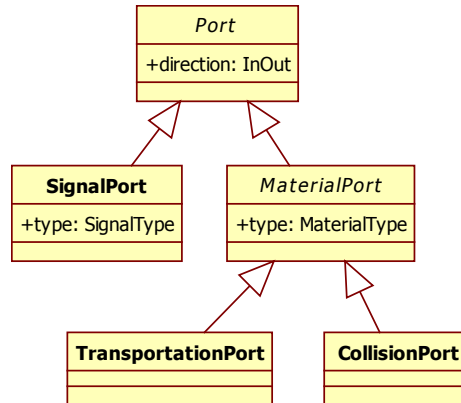
Fig. 3. The different port types supported

the component. To describe the type of signals exchanged on a signal port, we can use any classical type system (*e. g.,* algebraic specifications [EM85]), as long as it provides a carrier set for each type. To be practically applicable at least the basic primitive types (boolean, integral, and floating point values), enumerations, and tuple types should be supported, as well as the usually operations on them.

The remaining ports deal with material, which exists in the physical world and thus these ports 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 copied, each 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 such objects is straightforward. Collision ports are used to explicitly describe which collisions are allowed, which is described in 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 channels of the components for communication are proposed. The automaton model used here is influenced mostly by the one used in AutoFocus [BHS99] and extended using ideas from [Hen00]. An overview diagram is given in Figure 4.

The state space of such an automaton is described by control states, from which one is the initial state, and variables having a type and an initial value.
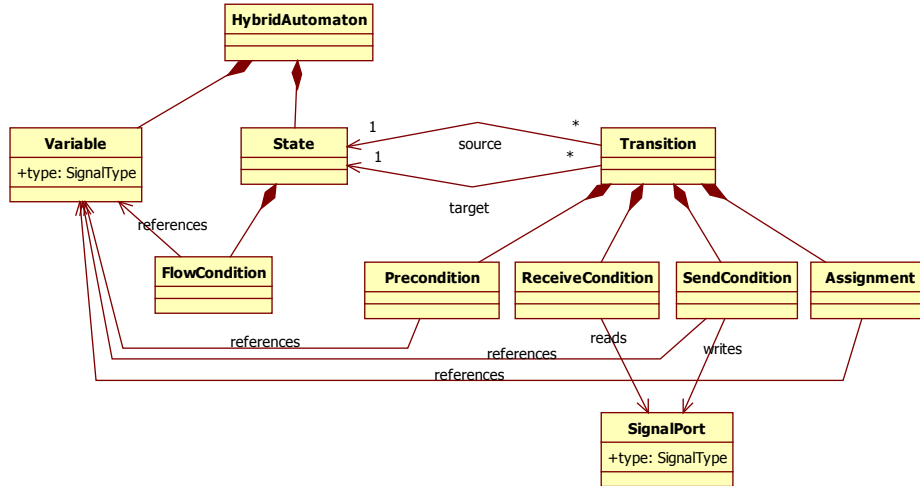
Fig. 4. The basic automaton model used

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.

Control states are connected by directed transitions, which describe the control flow of the automaton. A transition is a labeled arc having a source and a target state and a priority. Each transition can have receive conditions, a precondition, send commands, and assignments (post-conditions). Receive conditions describe, which values have to be present at the input signal ports, or are used to bind them to variables, while the precondition is a Boolean expression over the variables (both those of the automaton and from the receive conditions). A transition is called *active*, if both its receive conditions and precondition are fulfilled. If a transition is active, it may be executed, which consists in sending values according to the description given in the send commands and assigning new values to the variables according to the assignments, which may both be based on the current values of variables.

The expressions supported by the automaton mostly depend on an implementation of an editor or simulator and thus are not given here. Usually the basic Boolean operators and general arithmetic on the types used are supported. The automata described here, only deal with the signal ports. However there are some extensions described in the following sections which make them also define behavior for material and material ports.

3.5 Geometry and Motion

To support the visualization of the machine and the detection of collisions between machine parts or material with parts of the machine, geometric information is incorporated into the model. To us the geometry is just a compact subset

(*i. e.,* bounded and closed) of three-dimensional space. One possible model which is just composed of geometric primitives is shown in Figure 5, however more complex geometry models including arbitrary shapes could be used as well, provided there are algorithms for testing them for collisions.
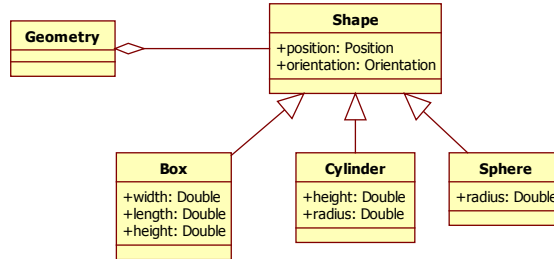

Fig. 5. A simple model used for describing geometry

All physical components are assigned a position, an orientation, and a geometry. Additionally each component can be augmented by at most one motion axis describing one degree of freedom (either 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. Minimal and maximal values are available to the automaton as constants to implement implicit end contacts.

For modeling more complicated movements, a motion hierarchy is introduced. 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 the children in the motion hierarchy. For some cases (such as the modeling of conveyor belts) we want components whose movement only affects children (and material as described in the next section) but not the component itself. We call such components *static*. Note that 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 only seen during simulation. 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 a position and an orientation. The geometry is given by the material type. Material objects are always managed by exactly one physical component, which we call the *owner* of the material object.

The life cycle of material objects is controlled by the component which currently owns it. There are *new* and *delete* operations, which can be used for creating new material or delete owned material in a transition of the component's automaton. The *new* operation takes the type of material created, the initial position and orientation given as an offset to the position of the creating component, and additional parameters defined by the material type (constructor call). The delete call gets the set of components to delete as a parameter. Those sets are defined using predicates, which are evaluated on all owned material objects. This way material can be selected based on its type, some property, or its location. To simplify the definition of location, a component can define *locators*, which are geometric objects with a position relative to the component (*i. e.,* they follow every movement). The predicates then may select material objects contained in or 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 then 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. However a component may still reject new material by means of *acceptance filters*, which describe 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 is an error condition, *i. e.,* the machine does not conform to its specification. All extensions introduced in this section are summarized in Figure 6.

3.7 Collisions and Collision Response

As stated in Section 2, collisions should be explicitly treated in our model. The detection of collisions is the task of the execution environment (simulator or verifier), as collisions are just treated as incoming events. The response to such collisions however depends on several factors and as such has to be modeled explicitly.
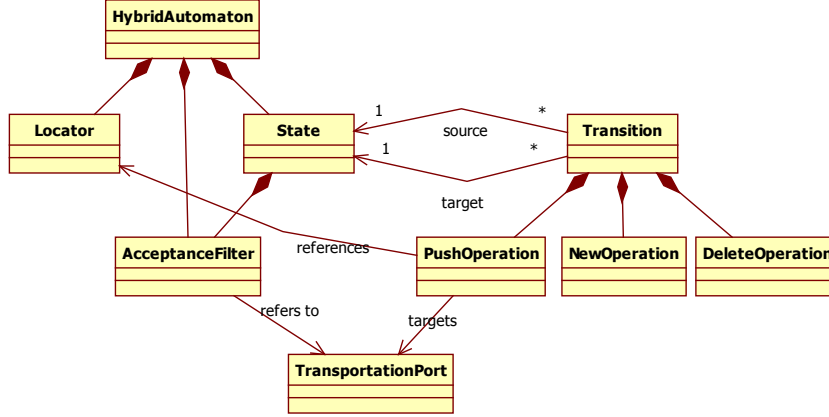
Fig. 6. The extended automaton model for dealing with material. Those parts already shown in Figure 4 are nor repeated here.

There are three different cases of collisions. First are collisions between two material objects, which cause a material object to be stopped if its current path of motion is blocked. Additionally the owners of the material objects are queried, if material managed by them may collide. If any of them disallows collisions, this is an error condition, which halts the simulation (or for verification purposes we would try to ensure that no error condition occurs). The answer of the components depends on a collision flag set for each component, which can be overridden by an additional per state flag. Using these flags one can model parts of the machine where collisions are allowed (*e. g.,* queuing up the material) or disallowed as they might lead to damage or congestion.

The second case are collisions between a material object and a physical component. If there is a channel between the component owning the material object 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 component blocks its way and is solid. Collisions of material with its owning component are always ignored.

Finally for collisions between physical components the reaction depends on the solidity of the components. If both components are solid, this is an error condition unless one is the motion predecessor of the other one in the motion hierarchy. This exception exists as we generally expect such collisions to be harmful to the machine, unless they are joined to move in conjunction. If none of the components is solid, the collision is ignored. For collisions between solid and non-solid components, the situation is similar to collisions between material and components. 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*, which is only used for this purpose. Then the collision is reported on the corresponding collision port as expected.

3.8 Discrete Simulation

This section describes how a discrete simulation of the model with fixed time steps works. While this is not a substitute for formally specified semantics, it can act as a starting point for it. Additionally the simulation description wraps up all dynamic aspects of the model described before, to set them into context of each other.

The simulation is initialized by setting all variables to their respective initial values and making the initial state of each automaton its current state. No material is initially in the model. The execution process then consists of the following steps. As we are simulating with a fixed time step, this value Δt has to be determined in advance.

1. **Propagate signals**
   Clear the values present at all signal input ports and copy the values from the corresponding signal output ports there. Afterwards clear those output ports. Only ports of components having an automaton defined are considered here. Components acting as containers are just "tunneled" through.

2. **Check for collisions**
   Based on the current positions and orientations of physical components and material objects, collisions are calculated. Based on the rules given in Section 3.7 collision events are generated and assigned to the collision ports of the components. If a collision leads to an error condition, the simulation halts.

3. **Discrete step**
   Each automaton performs a single step. For this all transitions leaving the current state are considered. If there are no active transitions, nothing happens. Otherwise the active transition with the lowest priority is executed. If there is more than one active transition having lowest priority, it is undefined which one is executed (non-determinism). The execution of the transition sets the current state of the automaton to the target state of the transition. Additionally the values of variables might be updated and new values written to the output ports. The transition may depend on the values of signal input ports and collision ports and can assign values to signal output ports and material objects (via *push*) to transportation output ports.

4. **Hand over material**
   For material located in a transportation output port (from a push in the previous step) the connected transportation input port is determined. The component providing this port becomes the new owner of the material objects unless the acceptance filter rejects the material objects. In this case the simulation halts with an error.

5. **Continuous step**
   For each automaton in the model the flow conditions of the current state are evaluated, and the affected variables are updated accordingly.

13

6. **Move components and material**

    The discrete and continuous step may have changed the current position of a component, as this is made available as a variable to the automaton. Now these movements are applied to the model. For material objects the motion of the owning component is applied, unless this path is blocked (which is known from the collision data collected in step 2).

An important aspect arising from these steps is that each automaton operates on the input signals generated the step before. This behavior, which is inherited from FOCUS/AutoFOCUS, ensures that no causal loops occur, which would make any operation order invalid.

3.9 Time

The previous section used a model of discrete equidistant time. It is easily seen that different time steps can lead to different simulation results, as collisions might be detected later or missed completely and differential equations are evaluated more coarsely. The obvious solution would be to define the semantics using continuous time. This however makes a simulator or analysis slightly more complicated, and we do not yet know whether the differences between a continuous time model and discrete time with a small time step are relevant for practical purposes, so we delayed this issue until we have more experience with those models.

4 Examples

In this section we will give examples how various parts of an automation machine can be modeled using our approach. To provide readable 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 be symbols on the border of the components. The symbols we are using in here are summarized in Figure 7. All other elements, such as transition conditions, are given textual. We will not provide a concrete syntax for them, but hope that they are understandable from the examples.
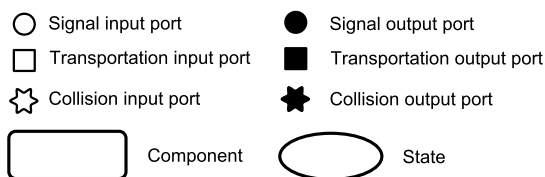

Fig. 7. Symbols used for components and ports

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 the source and the receiver, the receiver detects this and send corresponding signals. Thus this barrier is just a converter from collisions (with the light beam) to signals.
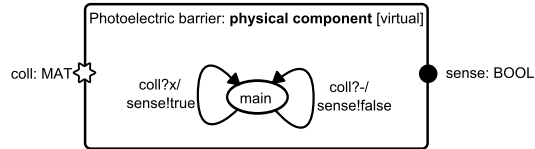

Fig. 8. Model of a photoelectric barrier

A possible model of a photoelectric barrier is given in Figure 8. The collision input port is of some type MAT and the signals sent are just 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 are thus not explicitly modeled. The automaton is very simple and consists of a single state. If some material collides and consequently can be "received" at the collision input port (`coll?x`) we send the value *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 some material MAT, which is shown in Figures 9 and 10. It has one signal input, which accepts a value used for setting the transportation speed. The automaton used is slightly more complicated than the one used before, although it still has just one state. In this state 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 of the conveyor.

The automaton has two transitions. The first one just pushes all material which intersects with the locator `loc_end` to the next component connected to `m_out`. This locator is also defined in the geometric view and is located at the end of the conveyor, handing all material objects leaving it to the next component. The second transition is active, if a new speed value is present at the `v_in` port. This value is stored in `v` and the material is pushed forward just as in the first transition.
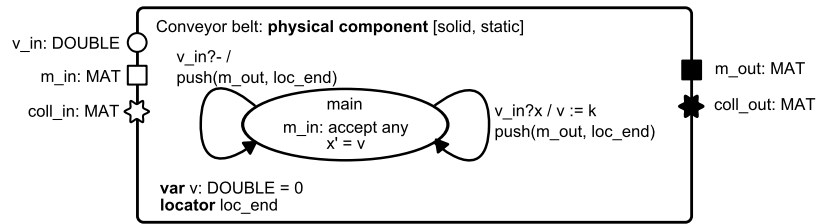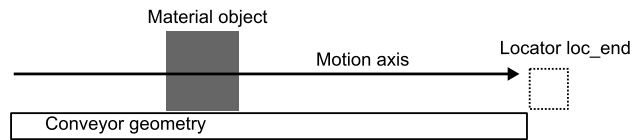
15

Fig. 9. Model of a conveyor belt

Fig. 10. Simplified 2d geometry of the conveyor belt

4.3 Grouping unit

Now we will compose a grouping unit from the components defined so far. The purpose of the grouping unit is to preprocess a stream of material objects with random distances between these objects and form a material stream, which consists of groups of n material objects with each group separated by a gap of given length.

One solution for this problem is to use two conveyor belts, which are positioned next to each other and whose speed is carefully controlled 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 the same speed until one more material object is handed over. The position of the material objects on both conveyors is located using photoelectric barriers. A possible realization is shown in Figure 11, where the conveyor belts and the photoelectric barriers are just the components described before. The controller is a computational component which adjusts the speed of the conveyors based on the signals from the photoelectric barriers and some configuration commands received via the third input port. Its specification is not shown here, as it is just a plain timed automaton. The entire grouping unit is a logical component and its ports just act as repeaters.

5 Related work

Our work was heavily influenced by the FOCUS modeling theory [BS01] whose primary purpose is the description of reactive systems. There also is a modeling
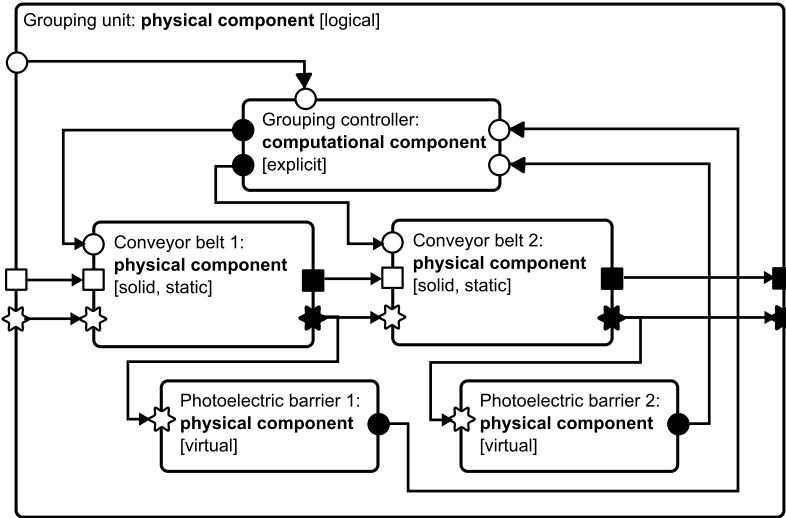
Fig. 11. Model of a grouping unit

language and a tool based on FOCUS, called AutoFOCUS [BHS99]. Actually the high-level model shown in Figure 1 is similar to the one of AutoFOCUS and if we eliminate physical components from our model (in conjunction with material ports) the remainder is quite similar to it. The notion of material, properties of material, geometry of components, collision detection and collision handling however are not directly supported. If not impossible, modeling the hardware aspects of automation machines would be really tedious.

The lack of support for geometry and collision handling also is the main reason to exclude all of the well-known traditional approaches for modeling systems, such as state charts [Har87], I/O automata [Lyn03], Petri nets [Pet62], process algebras such as CCS [Mil80] or CSP [Hoa85], or any of the many variants of them.

SysML (Systems Modeling Language) [Obj06] is a language for modeling systems by the Object Management Group (OMG). 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. Thus it would be rather complicated to apply formal analysis to these methods, besides the objections given for all other techniques before: missing support for material flow and collisions.

Machine simulation models, which are used for virtual commissioning, which is a software-in-the-loop test using the actual controller hardware and a simulated machine, are another possible source for modeling techniques. Existing industrial solutions however do not couple the geometry tightly to the components of the machine, but only describe reactions of a visualization model to state changes of the machine. This way collisions can only be observed by the model, if they are explicitly checked for (which usually includes the implementation of

a collision detector). For example the project SEMI[3] [ABH+99] used ROOM [SGW94] and C++ for modeling the simulated machine. Consequently support for continuous changes within the machine, a notion of material, or collisions as part of the machine logic were not available and had to be bypassed somehow by the modeler.

So far we have seen that the main problem of existing approaches is mostly in the lack of support for dealing with geometry, material, and collisions. Collision detection and response are one of the main areas treated by physical simulations. The usual drawback of those simulations is the missing support for discrete changes, which may be caused by collisions or signals sent by the controller. The modeling language Modelica [Til01] solves this problem by supporting both continuous flows and discrete changes (hybrid modeling). For collision detection and response an extension is proposed in [Eng00]. 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. Additionally, while collisions are treated physically correct, it is still not differentiated which collisions are allowed or disallowed. 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 Future Work

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. The discrete simulation might be insufficient for some cases, so it would be nice to define a continuous time interpretation of the model, or characterize the effects that can or can not be observed when using the time discrete simulation. Additionally communication between components always is discrete (*i. e.,* only single values and not functions over time are transmitted), while for some cases a real hybrid model (not only hybrid in local automata) might be useful. For both cases, however, we first have to gain more experience in modeling different kinds of real machines. This is also a precondition for extending the language from production machines to other sectors of automation machines, such as machine tools or machines from 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 the actual usage of the model, for which we might need a more formal semantics. These include the test-case generation and verification of properties, which were the driving applications for creating our model. However, there are other possible applications of such an integrated model, starting from discussing requirements with a customer or acting as the

---

[3] Simultaneous Engineering zur Entwicklung von Maschinen mit Mikrosystemen — Simultaneous Engineering for the Development of Machines with Micro-systems

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 some extensions of the model, such as annotations of fault behavior and error probability for components.

Finally we consider tool support for the presented technique to be an important issue, as only suitable tools allow to actually model and simulate real machines and better understand the implications of our design decisions. Furthermore the models used to perform testing, test-case generation, and verification have to be created in some way. There is already a proprietary editor and simulator for an early version of the modeling technique, but we plan to build a new editor for the revised system model described in this paper.

7 Conclusion

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.

As we rejected many existing modeling techniques because they did not fulfill the requirements we initially defined, we should also check to what extent our proposal does so. Requirements 1 (hierarchic components), 3 (explicit material support), and 5 (geometric data) are clearly fulfilled, but they are also the most easy ones as we just extended our modeling technique to include the required information. Requirement 2 (interfaces) is fulfilled in a sense that the syntactic interface is made explicit. However, the components are only compositional (*i. e.,* we can derive the behavior of the composed component from the behavior of the components it is composed of) if no error conditions occur (due to unexpected collisions). If the composed system is "correct", *i. e.,* there is no valid input leading the system into an error condition, the composition can be made compositional by choosing the right semantics. Requirement 4 (detection and handling of collisions) is realized by the collisions ports, which we consider a suitable solution to this problem. For the 6th requirement (abstraction) is depends somewhat on the point of view. The models we are using are often more abstract than those usually used, *e. g.,* for virtual commissioning, but for some applications an even higher degree of abstraction might be desired. Finally the last requirement (semantics) is currently our weak point. While the simulation step described in Section 3.8 is a first step in this direction, it is still far from formal semantics.

References

[ABH+99] 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*. Informationstechnik im Maschinenwesen. Herbert Utz Verlag, 1999. ISBN 3-89675-482-3.

[BHS99]   Manfred Broy, Franz Huber, and Bernhard Schätz. AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, 13(13):121–134, 1999.

[BS01]   Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer, 2001.

[Dij72]   Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[EM85]   Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*, volume 6 of *Monographs in Theoretical Computer Science. An EATCS Series.* Springer, 1985.

[Eng00]   Vadim Engelson. *Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing.* PhD thesis, Linköpings Universitet, 2000.

[Har87]   David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hen00]   T.A. Henzinger. The theory of hybrid automata. In M.K. Inan and R.P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, NATO ASI Series F: Computer and Systems Sciences 170, pages 265–292. Springer, 2000.

[Hoa85]   C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Lyn03]   Nancy A. Lynch. Input/output automata: Basic, timed, hybrid, probabilistic, dynamic, .. In *CONCUR 2003, 14th International Conference on Concurrency Theory*, volume 2761 of *Lecture Notes in Computer Science*, pages 187–188. Springer, 2003.

[Mil80]   Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer, 1980.

[Obj06]   Object Management Group. OMG SysML specification v. 1.0, May 2006.

[Pet62]   Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Universität Bonn, 1962.

[SGW94]   Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley and Sons, 1994.

[Til01]   Michael Tiller. *Introduction to Physical Modeling with Modelica.* The Springer International Series in Engineering and Computer Science. Springer, 2001.