Proc. GI workshop Rigorose Entwicklung software-intensiver Systeme, *Berlin, August 2000*

Heterogeneous Development of Hybrid Systems*

István Péter[1], Alexander Pretschner[2], and Thomas Stauner[2]

[1] Lehrstuhl für Informationstechnik im Maschinenwesen,
Technische Universität München, Boltzmannstr. 15, 85748 Garching, Germany
www.itm.tum.de/~IP
[2] Institut für Informatik,
Technische Universität München, Arcisstr. 21, 80290 München, Germany
www4.in.tum.de/~{pretschn,stauner}

Abstract. Traditional approaches to the development of mixed discrete - continuous, or hybrid, systems require an early partitioning of their models into discrete and continuous components. We present a development process that allows for postponing this partitioning decision. Different UML-RT based graphical description languages for different views on the models are discussed along the lines of an industrial case study that has been implemented within a CASE tool that allows for both modeling and simulating hybrid systems.

1 Introduction

The development of hybrid, i.e., mixed discrete and continuous, systems is an interdisciplinary task. Usually engineers from different disciplines are involved and must discuss their designs. Graphical description techniques provide a very useful means to support this communication. Just as for safety critical discrete systems, it is furthermore desirable to apply a high degree of mathematical rigor in the development of safety critical hybrid systems.

We argue that applying conventional SW development processes to hybrid systems results in a major drawback: These conventional processes require an early partitioning of the model into discrete and continuous subsystems which may result in a costly redesign of the whole model in later stages of the development.

The development process we advocate is supported by MaSiEd [1], a ROOM-based [20] CASE tool prototype for the development of hybrid systems that integrates description techniques for both discrete and continuous systems. These include architecture diagrams for the structural view as well as extended state machines and continuous block diagrams for the specification of the behavioral view on system components. The integration of HySCs [10], a hybrid variant of UML's Sequence Charts [19] into the tool, is the subject of ongoing work that allows for the description of use cases or exemplary component interactions.

MaSiEd differs from popular tools (e.g., Matrix_X/BetterState, Matlab/Simulink/StateFlow, Statemate/VisSim) in that its hybrid notations allow for an integrated

* This work was supported with funds of the Deutsche Forschunsgemeinschaft under reference numbers Be 1055/7-1, Be 1055/7-2 and Br 887/9 within the priority programs *KONDISK* and *Design and design methodology of embedded systems.*

development of hybrid systems. In Matlab/Simulink, for instance, components are either discrete or continuous rather than both. Their focus clearly is on continuous systems. Discrete switches from one continuous behavior to another (e.g., the different modes of friction) have to be modeled explicitly in the continuous (block diagram) as well as in corresponding discrete components. In particular, MaSiEd allows for the integration of continuous behavior into states rather than entire components. Furthermore, its underlying object oriented design principle supports re-use of components as well as the dynamic creation of system components (actors). These concepts are illustrated by parts of a large industrial case study, a wire stretching plant.

For the development of safety critical systems, the use of formal methods and notations is the prerequisite for mathematically proved assessments of a system's properties (e.g., model checking for discrete systems, or the determination of Eigenvalues for stability analysis in the case of continuous systems). For mixed discrete and continuous systems, there are only very few verification techniques yet (e.g., HyTech [3] which is not suited for large scale applications). For the applicability of future integrated verification techniques, a common formal semantics is mandatory.

A development process that is based on a notion of iteratively refining a system's design, i.e., transitioning from more abstract to more concrete models, also profits from a formal semantics. Checking consistency with a more abstract model – the model available before a refinement step – requires the definition of correct refinement relations [5]. Determining the correctness of such relations is impossible without a formal semantics. The definition of such a formal semantics is the subject of ongoing work [17].

2 A Development Process for Hybrid Systems

In this section we outline how conventional SW development processes are applied to the design and specification of hybrid systems. We discuss advantages and drawbacks of this approach, and present a more visionary approach preventing these drawbacks that is supported by the CASE tool MaSiEd. It results from carrying over ideas like graphical specification with different systems views and model based validation based on formal methods to hybrid systems. A central characteristic of the proposed approach is that it is based on notations that have a clearly defined semantics.

A conventional development process. A conventional development process for hybrid systems builds upon isolated description techniques for purely discrete and purely continuous components. Popular in industry are tool couplings such as using Statemate together with Matrix_X or the MATLAB/Simulink/StateFlow environment [9, 7]. For the development of safety critical systems we advocate the use of formal methods and notations wherever possible. This hinders the use of current commercial tools such as Statemate, ObjectGeode, Rational RoseRT or Stateflow. Their notations only have a formal syntax, but *the semantics remains imprecise and ambiguous, or very complex.* A semantics for the coupling with continuous tools in not defined anyway. For continuous systems there also exist analysis and simulation tools based on block diagram notations, e.g. MATLAB [22]. Note that we regard block diagram descriptions of continuous systems as formal here, because a mathematical model can be associated with individual blocks and their interconnection in a straightforward

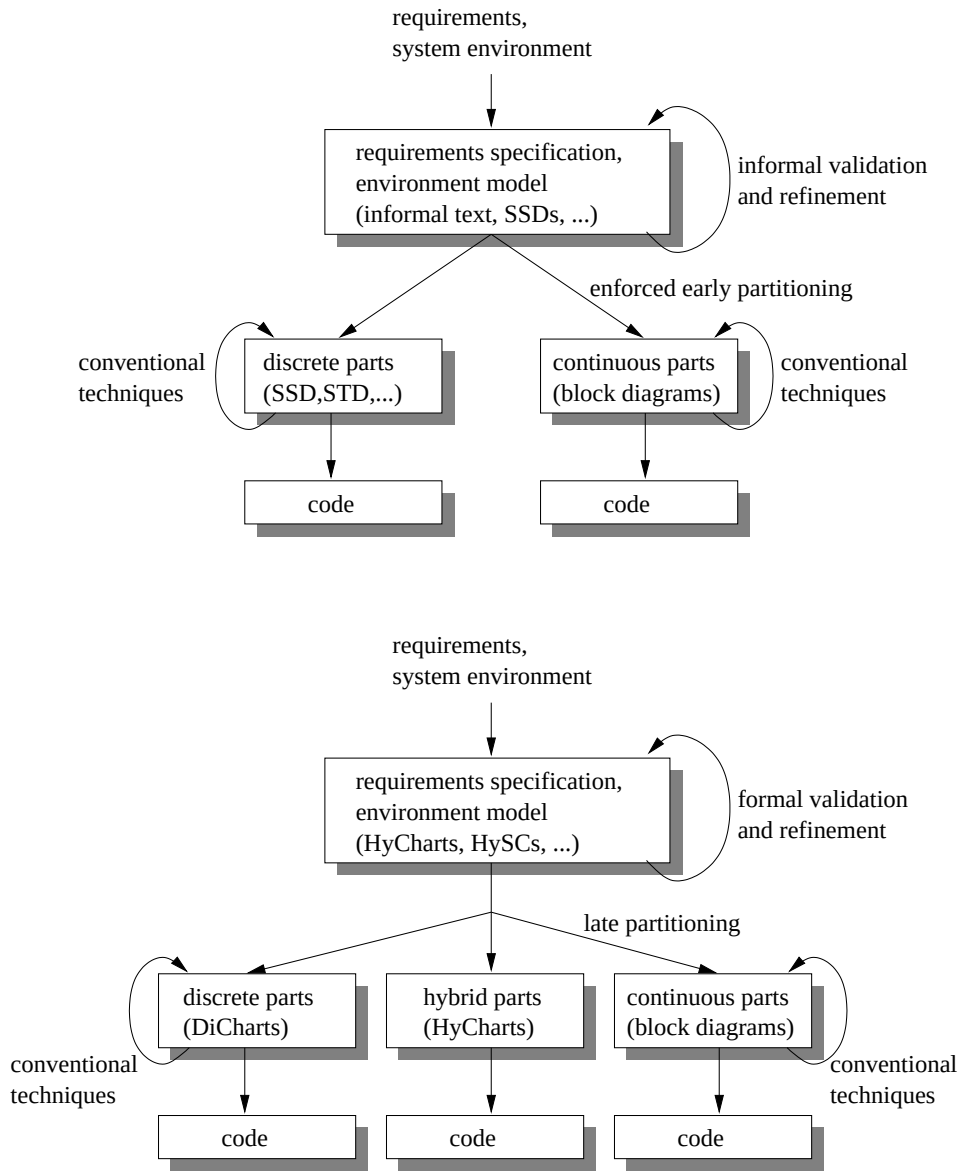

Fig. 1. A conventional development process (top) and an integrated development process with hybrid description techniques (bottom).

manner. (Nevertheless, the user has to keep in mind that the selection of integration algorithms for simulation can have a great impact on simulation results and can cause them to differ strongly from the mathematical model.) As soon as the system under development is partitioned into discrete and continuous parts, a formal specification can be written down using these existing tools, see Figure 1, top. Well-known tech-

niques from the discrete and continuous world can then be applied to the respective parts of the model. For instance, model checking and automatic test case generation may be used for the discrete part and analysis of eigenvalues for the continuous part.

Drawbacks. So far, the only currently available technique for examining properties of the mixed system is simulation. There are hardly any analytical methods regarding the mixed model and there are no techniques which support design modifications that affect both parts of the model. In fact such modifications could necessitate a redesign of the whole model.

Furthermore, in such a development process a designer has to perform a number of development steps informally, i.e., without documenting them with clearly defined notations, before a clearly documented process can start, i.e., a process relying on formal description techniques. In particular, these steps include partitioning the design into discrete and continuous parts which may involve an (implicit) discretization of some parts. This is unsatisfactory since the partitioning decisions may be difficult to alter later on. Apart from that, the resulting coupled discrete continuous model often is not natural for some components of a hybrid system. For example, analog to digital (AD) and digital to analog (DA) converters, and in some systems the environment, are inherently hybrid components.

Outlook: An integrated development process. In a development process with hybrid description techniques, such as the one depicted in Figure 1, bottom, the designer is able to formally specify mixed discrete continuous models at early stages of the development process. In the context of formal methods, we refer to "refinement" as altering (or augmenting) a system's functionality without violating properties that have already been established. If validation and transformation techniques, such as simulation and refinement, are available for these description techniques, the model can be *systematically* designed to meet those system requirements which affect its discrete as well as its continuous aspects. Rudimentary versions of such techniques already exist and are an area of current research (e.g. [4, 8]). In later steps the model can be refined into discrete, continuous, and possibly some remaining hybrid submodels. For the discrete and continuous submodels conventional techniques can then be used to realize those properties which only affect the respective part. Thus, the availability of formal hybrid description techniques and supporting methods for them pushes the point at which systematic development, i.e. development with formal description techniques, can begin towards the beginning of the analysis phase. A partitioning into discrete and continuous submodels can be postponed towards subsequent development phases. Such a development process with hybrid description techniques allows to obtain greater confidence in the model before a partitioning. Namely, testing and model checking techniques can be used to analyze requirements and refinement techniques can be used to guarantee some requirements by construction. By postponing implementation related questions changing requirements can more easily be taken into account. Thus, errors made in the initial development phases can be found earlier and are therefore cheaper to correct.

The development process we propose in Figure 1, bottom, is based on description techniques developed within our group in the last years. For requirements specification and environment modeling it uses the MSC-like notation *HySC* [10], and the

combination of architecture diagrams and a hybrid automata variant which is subsumed in *HyCharts* [11]. A methodological transition from HySCs to HyCharts is ongoing work (for similar work on discrete systems see [15]). Succeeding steps in the figure refer to HyCharts rather than to HySCs. As notations for the discrete and the continuous part we propose DiCharts [12], a discrete-time variant of HyCharts, and (continuous time) block diagrams, repectively, taht can be integrated easily into the HyChart notation.

Note that the aspect of postponing the partitioning of a system into discrete and continuous parts is related to the area of hardware/software codesign [6]. There, the decision on which parts of a system are implemented in hardware and software is postponed to later phases. However, unlike hardware/software codesign the partitioning into discrete and continuous components proposed here does not yet imply how the components are implemented. The discrete part could be implemented in software or on digital hardware, the continuous part can be turned into a discrete-time model and implemented in software (or digital hardware), or it could be implemented in analog hardware.

While there is hardly any tool support for the integrated process today, a close coupling of discrete and continuous notations in the HyChart style is implemented in the *MaSiEd* tool [2], which also allows simulation. The *HyTech* tool [13] (or other tools, e.g., Uppaal or Chronos) which offers model checking of hybrid models is another element needed as support for an integrated development process. Presently, however, its application is limited due to scalability problems and deficits of the underlying hybrid automata model [16]. Promising tool approaches for the future should couple analysis algorithms like those implemented in *HyTech* with modular graphical description techniques, e.g. HyCharts, in comprehensive tool frameworks.

Note that the development process for hybrid system proposed in [7] can be regarded as an intermediary between the two processes outlined here. There, the authors propose to complement block diagrams and automata-based notations with formal specifications using Z [21].

3 MaSiEd (Machine Simulator/Editor)

In this section, we present a CASE tool that partially supports the above development process for hybrid systems. MaSiEd is a CASE tool for modeling, simulating and analyzing the I/O behavior of general discrete, continuous, and hybrid systems. More particularly, it has been tailored to the needs of modern (field bus based) manufacturing systems with the aim of testing the associated PLC (programmable logic controller) software. The possibility to create virtual machine models of manufacturing plants is a prerequisite for PLC tests. These tests are carried out during all phases of the machine development, in parallel with the mechanical construction using Simultaneous Engineering as guiding principle. The economic development of simulation software that assures signal compatibility at the interface with the PLC needed a special modeling language with support for (1) both event-discrete and time-continuous modeling, (2) efficient modeling capabilities by supporting the reuse of existing machine component libraries, and (3) acceptance in the machine manufacturer domain.

Modern manufacturing systems are compound systems with elements from different physical disciplines, e.g. mechanics, hydraulics, and electrical engineering, com-

bined with controllers. In order to obtain the required expressiveness of the modeling language, different methodologies have been combined.

3.1 Modeling discrete systems

The I/O behavior of modern manufacturing systems can be characterized as a mainly event driven discrete behavior (with incorporated continuous behaviors; the focus, however, is on discrete systems which decreases the adequacy of tools such as Matrix_X that focus mainly on continuous parts). The MaSiEd CASE tool enables one to model reactive systems using the real time object oriented modeling methodology (ROOM [20]). ROOM's emphasis is on the seamless use of models from the requirement/high-level design phase down to the low-level design and testing stages. ROOM plays as important role in the ongoing definition of the UML dialect for real time systems.

The primary concepts of the ROOM modeling language are: actors, protocols, ports, bindings, and ROOMcharts, and they are used to model architectures consisting of hierarchies of communicating concurrent components.

An actor is a concurrent active object that hides its implementation from other actors in its environment. Fig. 3, left, shows an architecture diagram where actors are depicted as boxes. An actor's interface consists of so-called ports through which it can communicate with other actors by exchanging messages. A message is a composite object consisting of a signal, and an optional message data attribute. Each port has an associated protocol that restricts the types of messages flowing through the interface. The type of an actor is defined by its interface (and the respective protocols) that appears on the actor's outside. Actors can be assembled into complex structures by interconnecting their ports with communication channels called bindings (Fig. 3, left). In a more abstract view, these networked structures can be interpreted as entity-relationship models. Such aggregations of actors always are encapsulated within a higher-level composite actor specification. The ability to encapsulate actor structures with a containing actor means that an actor can be used as an abstraction facility that replaces the underlying aggregates by a conceptual unit. Communication is also feasible via so-called Service Access Points (SAPs) that do not have to be connected to other components with channels. This is helpful if components like PLCs have hundreds of ports that would clutter an architecture diagram. In order to model dynamic structures, ROOM allows for dynamically creating and destroying actors and their bindings at run-time.

At least bottom-level actors are associated with a behavior. The latter can initiate activities by sending messages as well as responding to external messages. The behavior of actors in ROOM is specified by a variant of the statechart [23] formalism called ROOMcharts. ROOMcharts basically are extended state machines with hierarchic states, but unlike statecharts without parallel composition of states: Parallel composition is defined using architecture diagrams like in Fig. 3, left. This formalism can model asynchronous event driven real-time systems. The behavior of an actor is always in one of two modes: it is either waiting for an event to occur or it is busy processing an event. All events are represented by the arrival of messages. When an event is received, it may cause a transition of the behavior from one state to another. While executing the transition the behavior may undertake a set of detail-level actions, including sending messages to other actors. A transition that has been ini-

tiated is guaranteed to complete even if further events occur while it is in progress. This "run-to-completion" processing model significantly simplifies the specification of behavior.

With the exception of initial transitions that can be used whenever a new behavior must be initialized, all other transitions are triggered by events. The trigger specification of a transition may include an optional guard condition that can be used to refine triggering specifications. The order in which triggers are evaluated starts with the innermost state in the current state context. If no transitions are found at this level, the next hierarchic level is searched and so on until either a transition is triggered or the top level is exhausted. If no trigger is satisfied in this search process, the event simply is ignored and will have no effect on the system's state.

For obvious reasons, time plays a major role in real-time systems. Two general situations pertaining to time are of interest: the expiration of some time interval and the occurrence of a set moment in time. In order to adapt them to the ROOM behavioral model, these types of events are converted into messages. The timing facilities are provided in ROOM as shared service of the ROOM virtual machine.

As the development progresses, more details have to be added. Detail level specifications occur in the form of action code in transitions of ROOMcharts. This level of detail is handled quite adequately by the C++ programming language, which is used as fine level specification language in MaSiEd.

One of the major disadvantages using ROOM to model the I/O behavior of manufacturing plants is the impossibility to adequately describe the behavior of continuous components of a machine.

3.2 Modeling Continuous and Hybrid Systems

Even though the I/O behavior of most modern manufacturing systems can be mainly characterized as an event driven discrete behavior, there are also subparts which have to be modeled in a continuous/hybrid manner.

The primary concepts added to ROOM in order to obtain the hybrid ROOM (HyROOM) modeling language are: block diagrams, stores, and state activities. These concepts can be used to model hierarchies of communicating concurrent hybrid components. In order to support the modeling of continuous subsystems we adopted the block diagram notation (e.g., Fig. 5) used in control theory. The block diagram notation is a well known formalism for modeling, simulating, and analyzing dynamic systems. Block diagrams basically represent sets of differential equations. In MaSiEd, differential equations and difference equations can be formulated graphically in a hierarchical manner using drag and drop operations. It is possible to use a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors, and one can also customize and create his own blocks. Note that block diagrams are, among other things, a means for architectural specifications of continuous systems. After defining the model, one can simulate it, using a choice of fixed step integration methods such as Runge-Kutta or Dormand-Prince. In order to make the modeling of hybrid systems possible, we extended ROOMcharts, a dialect of statecharts [23] with the concept of continuous activities. Fig. 3, right, shows such an extended automaton. An ad hoc way of enabling control-loop behavior modeling is to specify a state's activity in the form of block diagrams (Fig. 5). Variables assigned to connectors in

the block diagram associated to the activity can be evaluated in the transition conditions belonging to the respective state. Numerical algorithms associated with the block diagram will stop execution upon exiting from the state. Different actors in a model may be multi rate, i.e., updated at different rates.

The newly introduced concept of a store, represented by a filled rectangle, enables the transfer of real valued message data from state machines to block diagrams. The last message arriving in a store can serve as input to a block diagram. Stores may be connected to other actors with input for continuous or hybrid behavior or analog outputs to external hardware.

3.3 Modeling and Simulation Infrastructure

MaSiEd provides a user-friendly graphical design interface where hierarchical block diagrams and ROOM models with inheritance can be edited in the same environment. Inheritance on both the structural and behavioral level provides a basis for reuse. It is also possible in the same modeling environment to capture the system requirements using HySCs (hybrid Sequence Charts, e.g., Fig. 4) and later to use the captured requirements for validating the model. Model data are stored in a rapidly accessible repository. MaSiEd includes an incremental model compiler to translate HyROOM models into C++ source code programs that are then compiled to run on a ROOM virtual machine. A DDE (dynamic data exchange) interface to Matlab/Simulink enables the use of an automatic C program segment generation based on Matlab Real-Time Workshop and the evaluation of the continuous models in the early stages of the development. The C code corresponding to the block diagrams translated by the Matlab Real-Time Workshop and the C++-code generated from the rest of the model are combined automatically without user intervention. The generated model-specific code uses various ROOM run-time services and is linked with pre-compiled Run-Time System libraries (MicroRTS, developed by ObjecTime Ltd.). Once the model compiled, it can be downloaded from the developing environment to a target computer running the VxWorks (or RTLinux) real-time operating system.

3.4 Example: Wire stretching plant

The subject of this paragraph is the description of parts of an industrial case study carried out at the first author's affiliation. We chose to include it here in order to show how different description elements – architecture diagrams, extended state machines, continuous block diagrams, and hybrid Sequence Charts – can be connected within the MaSiEd tool. The executable model of this section is used for real time simulation, debugging, and testing.

The system in question is a wire stretching plant, and its main purpose is to wind wire of different thicknesses on reels. The case study was done in order to test the discrete process control; the actual PLC has been connected to MaSiEd for this purpose. When the case study was carried out, no continuous activities could be directly modeled within MaSiEd; the original model consisted of a discretized version (where the discretization was done by hand during the modeling process). This was sufficient because, as stated above, the study's purpose was mainly to test some discrete components of the PLC software. The work described in this paragraph is a continuation

of these efforts in order to incorporate continuous behavior that can be used to test other parts of the PLC more adequately. After briefly explaining the overall system, we will concentrate on a specific part of it that has been re-modeled in a hybrid manner (i.e., including modeling concepts such as block diagrams. For the original study, these continuous aspects have been ignored (or, by ad hoc discretization, abstracted in a very rough manner). The system has been simulated in a multi media manner for PLC testing and for understanding its operational properties.

Structure. The wire plant's overall structure is as follows. The environment produces wire that enters the system at a variable speed. This wire has to be wound up on a reel. The turning reel's velocity has to be almost equal to the incoming wire's velocity in order to guarantee a homogeneously wound wire. This indeed is one of the main quality requirements. It's velocity is controlled by a device between reel and environment, called the *dancer*, that consists of a set of pulleys the wire runs over (Fig. 2).

Not all of the pulleys are fixed so that the wire's velocity is dependent on the vertical position of the loose pulleys in this device (in a sense, it is comparable to a hoist where loose pulleys can move up and down). The position of these loose pulleys is a measurable magnitude that allows to deduce the wire's speed *behind* the dancer, when its front is the part that is closest to the environment, i.e., the wire's original source.

Once a reel is totally wound up it has to exit the system. This is achieved by a table that brings a new (empty) reel in position after the full one has been put on a belt by this very table. This is a complex, mostly discrete process that involves moving the table, fixing the new wheel on the motor's axis, cutting the wire, and making the new reel turn. There are two main conveyor belts involved in the system, one for empty, and one for wound up reels. This part of the system was the focus of the original case study and is omitted here for brevity's sake.

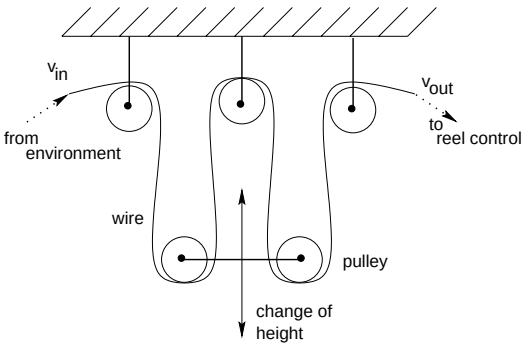

Fig. 2: Dancer

In addition to hydraulic aggregates that guarantee the fixed position of a (turning) reel on the axis of the associated motor – the very motor that interacts with the dancer via a controller for the winding speed –, the last main component of this system is the PLC part with roughly 180 I/O ports. We also omit these two components here for the sake of brevity.

Hybrid subsystem. This paragraph's focus is on the hybrid subsystem that consists of the dancer, the DC motor for driving the reel, and the controller connecting the DC motor with the dancer.

Its basic structure is depicted in Fig. 3, left, where continuous ports are marked with a semi-circle around a box. The systems input is the wire's continuously changing

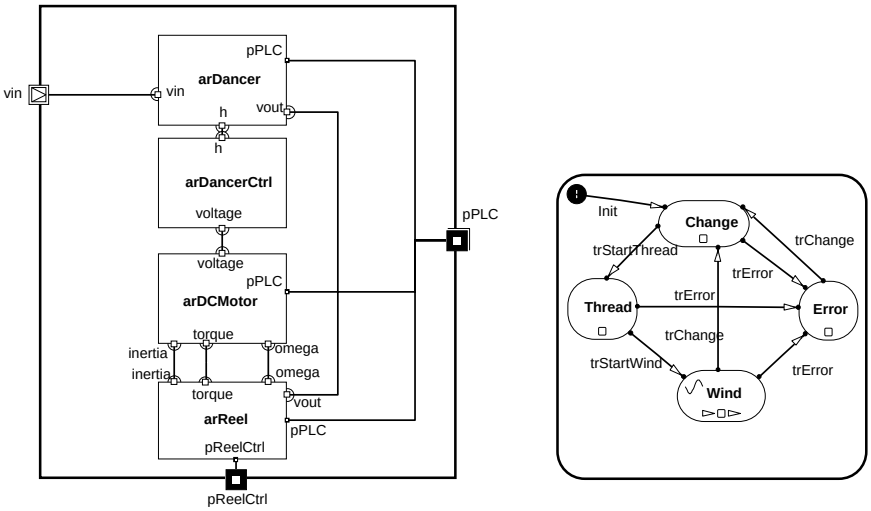

Fig. 3. Hybrid subsystem's architecture and reel's behavior

input velocity, v_{in}. The system communicates discretely with the PLC via port *pPLC*, and with the reel control via port *pReelCtrl*. The reel control takes care of exchanging a full reel in the system by an empty one.

Fig. 4 contains a hybrid Sequence Chart (HySC [10]) depicting a typical use case for this system. Hybrid Sequence Charts are a variant of UML's Sequence Charts [19] and use the standard Message Sequence Charts (MSCs [14]) notation. Sequence Chart dialects are a popular means of specifying use cases. They contain several axes each of which corresponds to one component. Time progresses from top to bottom, and a Sequence Chart shows one (incomplete) communication schema between the involved components. Communication is achieved with the help of events that are depicted by arrows from one axis to another. Unlike MSCs, HySCs employ a synchronous time model. Furthermore, they use the MSC condition boxes (depicted as hexagons) to refer to the (qualitative) state of one or more components. Dotted parts of an axis indicate that the associated signals occur simultaneously. In this figure, a use case for the normal operation mode is specified: First, an empty reel has to be inserted in the system (states *change*). Once the change is done, the *threading* process starts; the wire is put onto the new reel, and it is cut from the old one. If this process successfully completes, the actual *winding* process is initiated; compared with the *change* state, its main characteristic is a relatively high velocity of the reel. When the reel is coiled up, the PLC re-initiates the process of *changing* the reel by moving the full one out of the system and bringing an empty one in position. This overall process should be repeated perpetually. Note that this is just one use case where the possible existence of errors has been ignored. For the sake of brevity, we also omit the predicates that describe the states.

There are three states describing the dancer's behavior (actually, these three states are the main states of the whole subsystem). It can either be in a *winding state*

where the wire's output velocity, $v_{out}(t)$ should be controlled to be equal to its input velocity, $v_{in}(t)$. The change of the loose pulleys' height, h, makes the dancer contain more or less wire, and it thus acts as a buffer the inertia of which is needed for controlling the reel's angular speed. Remember that the task is to wind the wire in a homogeneous manner - this can be achieved if the reel rotates as fast as the wire enters the system. In this state the relationship between h, v_{in}, and v_{out} is given by $h(t) = \int_{t_0}^{t} \frac{v_{in}(t)-v_{out}(t)}{2 \cdot n_{pulleys}} \mathrm{d}t + h_0$ where $n_{pulleys}$ denotes the number of loose pulleys in the system.

If the reel is full and has to be exchanged, the wire has to move at a very slow speed, v_{min} (it actually never really stops). This is achieved by moving the loose pulleys downwards, and the dancer is being filled up with wire.
This state, *threading*, comprises (a) ejecting the full reel, (b) cutting the wire, (c) moving an empty reel in position, and (d) threading the wire into this new reel. Its continuous behavior is described by $h(t) = \int_{t_0}^{t} \frac{v_{min}-v_{out}(t)}{2 \cdot n_{pulleys}} \mathrm{d}t + h_0$.
The third state, *error*, is reached if the pulleys cannot move further because they have reached their limits, $h_0 \pm h_{max}$. Once *error* is entered $h(t)$ becomes immaterial since the normal operation mode is stopped. The model therefore simply leaves $h(t)$ constant. Note that the dancer's ROOMchart has not been depicted here; it resembles the reel's one (Fig. 3, left). The fact that many components in a hybrid system share the same structure (in terms of states) seems to be a common feature of ROOM based modeling (e.g., [18]), not only for hybrid systems. We found that ROOM's inheritance mechanism is helpful here for it allows one not to re-draw almost the same states for each component of system. In this case, we chose a top level behavior for the whole subsystem with three states, and this schema has been refined for the reel where state *change* is decomposed into two states, *change* and *thread*. The dancer's controller, on the other hand, consists of the same three states as the dancer itself. For the sake of clarity, we chose to describe a rather "flat" model here.

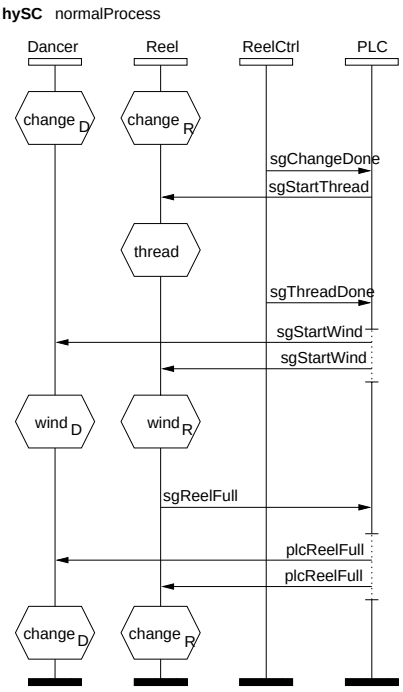

Fig. 4: Use case: normal operation

The last hybrid component of interest is the reel itself. Given the wire's input velocity, it keeps track of the reel's inertia, its torque, and its continuously growing radius (wire is being wound up), $R(t)$. It also yields the wire's output speed described by the algebraic constraint $v_{out}(t) = R(t) \cdot \omega(t)$. $v_{out}(t)$ is fed back into the dancer.

If the reel is turning, its radius changes according to $R(t) = \int_{t_0}^{t} c \cdot \omega(t) \mathrm{d}t + R_0$ where c is a factor determined by the wire's physical properties. Fig. 5 shows a Simulink block diagram for this formula where constant c has been replaced by its actual value that results from various physical properties. It is associated with state *wind* in Fig. 3, right. State associated with continuous activity are marked with a sine symbol in the

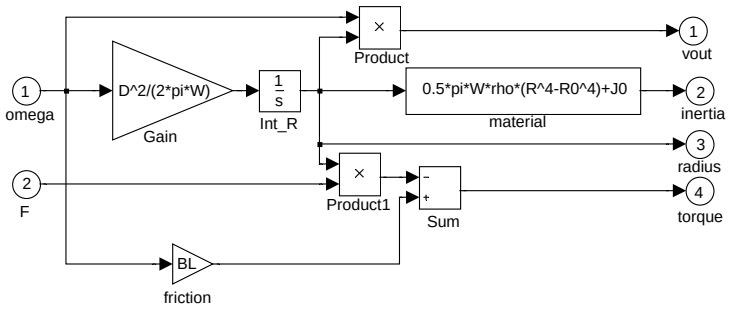

Fig. 5. Reel's dynamic in state *wind*

extended state machine's diagram. The integration takes place in the block labeled $\frac{1}{s}$. F is the force the wire applies to the reel, and B_L is a friction constant.

If the reel's radius changes, its mass also does. This is why in addition to the radius there are two further outputs for torque and inertia; these values are inputs to the motor's controller.

The reel can find itself in four different states: *wind*, *change*, *thread*, and *error* (Fig. 3, right). The reel (or connected sensors, respectively) tells component *reel control* when it is full; the control then initiates actions necessary for changing the reel and threading the wire into an empty one. It also tells the dancer about this change of state (via the PLC, causing a switch from one differential equation to another).

4 Conclusions

There is an increasing need for appropriate description techniques for hybrid systems. We have argued that simply glueing concepts from continuous modeling to discrete modeling or vice-versa results in a disadvantegeous development process: Its main drawback is the need for an unappropriately early partitioning of system models into discrete and continuous submodels. A development process that does not suffer from this major drawback has been presented together with (1) a combined notation, HyROOM, and (2) a CASE tool that partially supports it. This UML-RT based CASE tool supports different views on a model, including architecture, behavior, data, and interaction views.

Thus far, the only scalable validation technique for hybrid systems consists of simulation, or testing, respectively. The definition of a denotational formal semantics [17] for HyROOM may result in a possible connection of hybrid model checkers (e.g., HyTech, Uppaal, Kronos) to MaSiEd. It is also vital in the formal definition of refinement relations between models of different degree of abstraction at different stages of the development process. The existing semantics is complete with the exception of dynamically creating objects (specified by HySCs). The full integration of HySCs into MaSiEd is part of ongoing work. HySCs may be used for the automatic creation of automaton skeletons [15] as well as the specification of test cases for the (semi-) automatic generation of test cases.

In terms of modeling, ROOM based modeling turned out to be prone to copying state structures. This seems to be the case in particular for controller components that are connected to hybrid parts of the system. We found MaSiEd's design decision to support re-use helpful not only in this context. Finally, discussions often arose when it came to the question whether to model a state by a variable or a proper state (a circle in the behavioral view). Along side the tight integration of HySCs into MaSiEd and means to automatically generate test cases, a clarification of this last point in terms of design heuristics will be part of our future work.

References

1. J. Albert and Tomaszunas. Komponentenbasierte Modellbildung und Echtzeitsimulation kontinuierlich-diskreter Prozesse. In *Proc. VDI/VDE GMA-Kongreß Meß- und Automatisierungstechnik*, Ludwigsburg, Germany, 1998.
2. J. Albert and J. Tomaszunas. Komponentenbasierte Modellbildung und Echtzeitsimulation kontinuierlich-diskreter Prozesse. In *Proc. of VDI/VDE GMA Kongreß Meß- und Automatisierungstechnik*, 1998.
3. R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems I*, LNCS 736. Springer-Verlag, 1993.
4. M. S. Branicky. Stability of switched and hybrid systems. In *Proc. 33rd IEEE Conf. Decision and Control*, 1994.
5. M. Broy. A Logical Basis for Modular Systems Engineering. In *Calculational System Design*, volume 173 of *NATO Science Series F*. IOS Press, 1999.
6. K. Buchenrieder and J. Rozenblit. Codesign: An overview. In *Codesign – Computer-aided HW/SW Engineering*. IEEE Press, 1995.
7. M. Conrad, M. Weber, and O. Müller. Towards a methodology for the design of hybrid systems in automotive electronics. In *Proc. of the International Symposium on Automotive Technology and Automation (ISATA'98)*, 1998.
8. DFG. Priority program KONDISK (analysis und synthesis of continuous-discrete systems). www.ifra.ing.tu-bs.de/kondisk/, 2000.
9. M. Fuchs, M. Eckrich, O. Müller, J. Philipps, and P. Scholz. Advanced design and validation techniques for electronic control units. In *Proc. of the International Congress of the Society of Automotive Engineers*. SAE International, 1998.
10. R. Grosu, I. Krüger, and T. Stauner. Hybrid Sequence Charts. In *Proc. of ISORC 2000*. IEEE, 2000.
11. R. Grosu, T. Stauner, and M. Broy. A modular visual model for hybrid systems. In *Proc. of FTRTFT'98*, LNCS 1486. Springer-Verlag, 1998.
12. R. Grosu, Gh. Ştefănescu, and M. Broy. Visual formalisms revisited. In *Proc. of Int. Conf. on Application of Concurrency to System Design (CSD'98)*, 1998.
13. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HyTech. In *TACAS 95: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1019. Springer-Verlag, 1995.
14. ITU. ITU-T Recommendation Z.120: Message Sequence Charts (MSC), November 1999.
15. I. Krüger. *Using MSCs for design and validation of distributed software components*. PhD thesis, Technische Universität München, 2000.
16. O. Müller and T. Stauner. Modelling and verification using linear hybrid automata - a case study. *Mathematical and Computer Modelling of Dynamical Systems*, 6(1):71–89, 2000.
17. I. Péter, A. Pretschner, and T. Stauner. ROOM for Hybrid Systems: A formal grasp, 2000. TU München, Internal report.

18. A. Pretschner, O. Slotosch, and T. Stauner. Developing Correct Safety Critical, Hybrid, Embedded Systems, 2000. *New Information Processing Techniques for Military Systems*, NATO Research. To appear.
19. Unified modeling language, version 1.1. Rational Software Corporation, 1997.
20. B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons Ltd, Chichester, 1994.
21. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.
22. The MathWorks Inc. MATLAB. `www.mathworks.com/ products/matlab/`, 2000.
23. M. von der Beeck. A comparison of statecharts variants. In *Proc. of FTRTFT'94*, LNCS 863. Springer-Verlag, 1994.