Approaching a Discrete-Continuous UML: Tool Support and Formalization*

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

[1]Institut für Informatik, TU München
Arcisstr. 21, 80290 München, Germany
[2]Lehrstuhl für Informationstechnik im Maschinenwesen, TU München
Boltzmannstr. 15, 85748 Garching, Germany
{stauner,pretschn}@in.tum.de, peter@itm.tum.de

Abstract: This paper presents HyROOM, a proposal for an extension of UML-like languages by continuous activities for the specification of mixed discrete-continuous, or hybrid, systems. It is implemented in a CASE tool prototype based on the Real-time Object Oriented Modeling methodology. All vital parts of HyROOM's operational simulation semantics are mapped into HyCharts, a formal framework for hybrid systems. All essential concepts are discussed along the lines of parts of an industrial case study, a wire stretching plant. The semantics is the basis for validation and refinement techniques.

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. We believe that corresponding CASE tools—that are based on a clearly formalized semantics but hide it from the engineer—are crucial in alleviating the task of modeling systems. Clearly, excellent engineers are able to implement each imaginable kind of system (and their qualification seems to be the key prerequisite for the implementation of development processes such as Extreme Programming [Be99]). However, CASE tools may be one way of allowing less excellent engineers to successfully cope with the same kind of problems.

MaSiEd. We present MaSiEd [AT98], a Rational Rose RealTime [Co01] (or ROOM [SGW94]) like CASE tool prototype for the development of hybrid systems which has been developed in various projects at the third author's institution and

*Supported with funds of the DFG under reference numbers Be 1055/7 and Br 887/9 within the programs *KONDISK* and *Design and design methodology of embedded systems.*

which integrates description techniques for both discrete and continuous systems. These include structure diagrams for the architectural view as well as extended state machines and continuous block diagrams for the specification of the behavioral view on system components. HySCs [GKS00], a hybrid variant of UML's Sequence Charts [Gr00] that allow for the description of use cases or exemplary component interactions, are being integrated into the tool.

MaSiEd differs from popular tools for the development of hybrid systems(e.g., Matrix_X/BetterState, Matlab/Simulink/StateFlow, Statemate/VisSim) in that its hybrid notations allow for an *integrated* 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., different modes of friction) have to be modeled explicitly in the continuous components (block diagram) as well as in corresponding discrete components (defined by state machines). In contrast, 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 (capsules). The paper illustrates how the different description elements are used in a large industrial case study, a wire stretching plant.

Semantics. 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 few verification techniques yet (e.g., HyTech [Al93]). 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 also profits from a formal semantics. Checking consistency of a refined model with a more abstract model can be based on the definition of correct refinement relations [Br99]. Determining the correctness of such relations is impossible without a formal semantics. Machine support for sound refactoring [Fo99] (of hybrid system models) also requires a formalized semantics. Furthermore, the relationship between continuous parts of the model and their discretized counterparts for simulation and implementation can only be established with such a clear semantics. In addition, tool couplings cannot be done without matching the respective semantics.

We advocate the use of formal descriptions starting already in early design phases (requirements capture). For hybrid systems this requires formal *hybrid* description techniques: The use of separate discrete and continuous notations would require a partitioning into discrete, continuous, and remaining hybrid subsystems that is not adequate in early development phases where abstract models are desired [PSS00]. However, as stated above, this is the case with existing tools or tool couplings which are not suitable for early phases but rather for system design; but even in systems design their use suffers from imprecise, ambiguous, and complex semantics. No formal semantics for tool couplings are given anyway. This is why we chose to complement the description of MaSiEd with a mapping of all

its vital parts into HyCharts [GSB98a], a framework for hybrid systems based on a denotational semantics.

Contributions. This paper's contributions are the presentation of a CASE tool for hybrid systems, MaSiEd, and the clear definition of a formal semantics for this tool. The semantics forms the basis for (automated) verification/validation techniques, for refactoring/refinement, and for establishing a relationship between a continuous model and its discretized refinement for simulation as well as implementation purposes.

Related work. Apart from the tools for hybrid systems already mentioned above, there is plenty of work on simulation packages for hybrid systems (see [Mo99] for an overview). Often these packages offer convenient, sometimes application specific, graphical description techniques, but usually no formal semantics is defined for them. There also are simulation tools with a strong formal background. These, however, put less emphasis on visual specification. The ongoing work on the *Charon* system is an exception here, as it targets formal, visual specification as well as simulation [Al00]. A first approach towards a hybrid version of Statecharts can be found in [KP92]. The operational semantics given there, however, does not fully support hierarchy. In particular inter-level transitions and hierarchic specification of continuous activities are not possible. More work on semantics for hybrid systems can be found in [Gr93]. Some very interesting approaches include [Ly96] and the popular hybrid automata [Al93] for which there is some tool support and also a visual notation. Unfortunately, their practical use strongly suffers from a lack of modularity [MS00]. A central issue of our work is the research for a convenient modeling methodology for hybrid systems which is suitable for practice and can be put on a formal basis. Therefore, most of the work cited above is complementary to our approach, either dealing with the modeling and simulation of hybrid systems, or with formal models for them. An exception is the work in the context of [FNW98] where UML's class diagrams are extended for hybrid systems and coupled with Z specifications.

Structure. The remainder of this paper is organized as follows. After describing the CASE tool itself, we illustrate the used description techniques by referring to parts of a wire stretching plant that has been modeled with MaSiEd. Finally, we describe the mapping from MaSiEd models into the formal HyChart notations. Please note that the paper assumes some familiarity with UML, Rational Rose RealTime or ROOM. Technical details of this work may be found in [SPP01].

2 MaSiEd (Machine Simulator/Editor)

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 (by simulating) the associated PLC (programmable logic controller) software. This tailoring becomes apparent w.r.t. two characteristics. First, comprehensive libraries for components occurring in manufacturing plants exist.

Second, the simulation infrastructure allows the coupling of the simulation with PLC hard- and software. The modeling features themselves are also applicable to other hybrid domains, but favor systems were the discrete complexity dominates.

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

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 paradigms of the UML and further ideas for real-time systems present in Rational Rose RealTime and ROOM [Co01, SGW94].

The primary constructs used in MaSiEd are: capsules, protocols, ports, connectors, and state machines, and they are used to model architectures consisting of hierarchies of communicating concurrent components. A capsule is a concurrent active object that hides its implementation from other capsules in its environment. Fig. 2, left, shows an architecture diagram where capsules are depicted as boxes. A capsule's interface consists of so-called ports through which it can communicate with other capsules. The type of a capsule is defined by its ports (and the respective protocols) that appear on the capsule's outside. Capsules can be assembled into complex structures by interconnecting their ports with communication channels called bindings (Fig. 2, left).

At least bottom-level capsules are associated with a behavior. They can initiate activities by sending messages as well as responding to external messages. Their behavior is specified by extended state machines with hierarchic states, but unlike Statecharts [Be94] without parallel composition of states: Parallel composition is defined using architecture diagrams like in Fig. 2, left. The behavior of a capsule 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 capsules.

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. As the development progresses, more details have to be added. Detail level specifications occur in the form of action code in transitions of state machines. The transition code as well as triggering events and condition are specified in C++ systan in MaSiEd. In order to be able to treat transitions between different hierarchic levels in the state machine in a modular way the concept of *transition points*, as introduced in [SGW94], is included in MaSiEd. They split interlevel transitions into segments such that each segment clearly pertains to exactly one hierarchic level.

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 are most adequately modeled in a continuous/hybrid manner. We therefore extended the notations adapted from Rational Rose RealTime by using the mechanism of UML stereotypes to obtain our *HyROOM* modeling language.

The primary concepts added are block diagrams 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. 2, bottom right) used in control theory. The block diagram notation is a widely used formalism for modeling, simulating, and analyzing dynamic systems. Continuous-time block diagrams basically represent systems of differential equations. In MaSiEd, differential equations and difference equations can be formulated graphically in a hierarchical manner using drag and drop operations (library support). Note that block diagrams are a means for architectural specifications of continuous systems. MaSiEd state machines are enhanced with the concept of continuous activities via these block diagrams: Fig. 2, right, shows such an extended automaton with a hierarchic state (OK) and a continuous activity in its substate WIND. Variables assigned to connectors in the block diagram associated to the activity can be evaluated in the transition conditions belonging to the respective state. Different capsules in a model may be multi rate, i.e., updated at different rates.

Modeling and simulation infrastructure. MaSiEd provides a graphical design interface where hierarchical block diagrams and HyROOM 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, [GKS00]) and later to use the captured requirements for validating the model. Model data are stored in a 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 [SGW94]. A DDE 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 fully automatically generated model-specific code uses various run-time services and is linked with precompiled Run-Time System libraries (MicroRTS, developed by ObjecTime Ltd.).

The compiled model can be downloaded from the developing environment to a target computer running VxWorks or RTLinux.

3 Example: Wire stretching plant

This paragraph describes parts of an industrial case study [PPS00] to illustrate the HyROOM language. The system in question is a wire stretching plant, and its main purpose is to wind wire 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. After briefly explaining the overall system, we concentrate on the part of it which has been modeled in a hybrid manner.

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. 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. 1). Not all of the pulleys are fixed so that the wire's velocity is dependent on the rate of change of the vertical position of the loose pulleys in this device. The position of these loose pulleys is a measurable magnitude that allows to deduce the wire's speed *behind* the dancer.

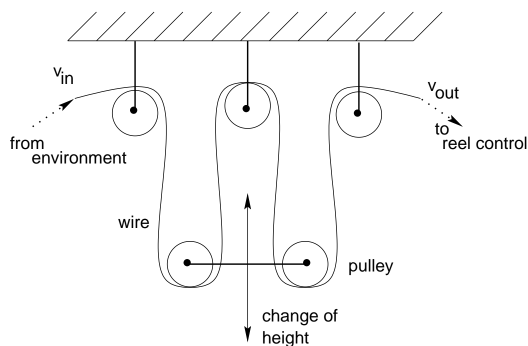

Figure 1: Dancer

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. This part of the system is omitted here for brevity's sake.

In addition to hydraulic aggregates that fix a (turning) reel on the axis of the associated motor the last main component of this system is the PLC part with roughly 180 I/O ports. Hydraulic aggregates and PLC are also not considered further here.

Hybrid subsystem. This paragraph's focus is on the hybrid subsystem which 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. 2, left. The system's input is the wire's continuously changing 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 for an empty one.

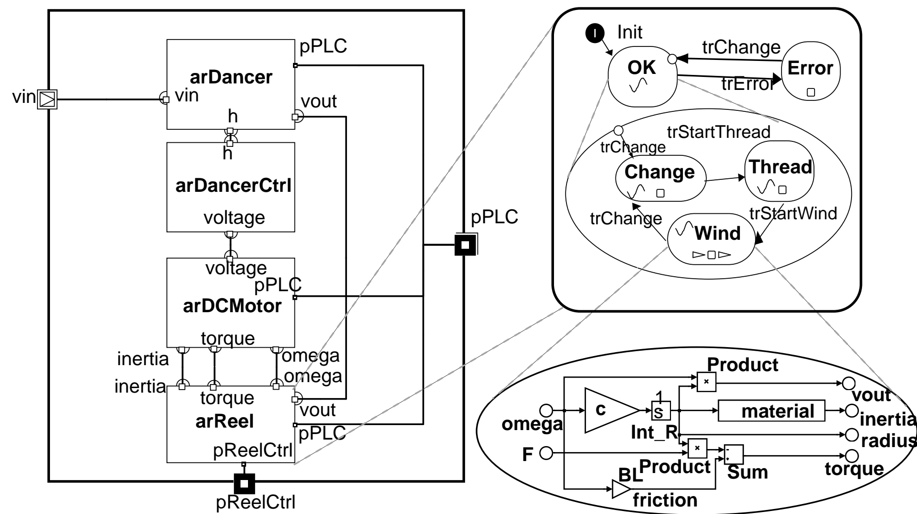

Figure 2: Hybrid subsystem's architecture and reel's behavior

A typical use case for the normal operation mode is as follows (space restrictions prevent us from graphically specifying it with a hybrid Sequence Chart [GKS00]; see [PPS00]): First, an empty reel has to be inserted in the system (state *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.

Besides an error state, there are two states describing the *dancer's* behavior (compare with hierarchic state OK in the *reel's* state machine in Fig. 2). 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 $h(t) = \int_{t_0}^{t} \frac{v_{in}(\tau)-v_{out}(\tau)}{2\cdot n_{pulleys}} \mathrm{d}\tau + h_0$ 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 (state *changing*). This state comprises (a) ejecting the full reel, (b) cutting the wire, and (c) moving an empty reel in position. Note that the dancer's state machine has not been depicted here; it resembles the reel's one

(Fig. 2, left). The fact that many components in a hybrid system share an almost identical control-flow structure (in terms of states) seems to be a common feature of modeling based on architecture diagrams (e.g., [PSS00]), not only for hybrid systems. We found that MaSiEd's inheritance mechanism is helpful here to avoid redrawing such similar model parts.

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 which is described by the algebraic constraint $v_{out}(t) = R(t) \cdot \omega(t)$ and fed back into the dancer. If the reel is turning, its radius changes according to $R(t) = \int_{t_0}^{t} c \cdot \omega(\tau) \mathrm{d}\tau + R_0$ where c is a factor determined by the wire's physical properties. Fig. 2, bottom right, shows a Simulink block diagram for these formulas. It is associated with state *wind* in Fig. 2, right. 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 and v_{out} there are two further outputs for torque and inertia.

4 HyCharts in a Nutshell

The modeling concepts underlying HyROOM are very similar to those introduced in the formal specification technique *HyCharts* [GSB98a, GS98]. Like HyROOM, HyCharts were invented as an adaption of concepts from the UML, Rational Rose RealTime and ROOM to hybrid systems. However, while the emphasis of HyROOM is to provide a tool environment for modeling and simulation, the work on HyCharts aims at providing a clear, modular formal semantics for the proposed hybrid description techniques. HyCharts consist of two graphical, modular description techniques. *HyACharts* (e.g., Fig. 3) are used for the specification of the system architecture and *HySCharts* for the specification of the behavior of a hybrid system's components. Very similar to HyROOM's state machines, HySCharts are an extension of ROOM-style hierarchic state machines by continuous activities that are associated with control-states. HyCharts regard a system as a network of components communicating over directed channels in a time-synchronous way.

Semantic domain. Mathematically, components Cmp are total relations on input and output trajectories, $Cmp \in \mathcal{I}^{\mathbb{R}_+} \rightarrow \mathcal{P}(\mathcal{O}^{\mathbb{R}_+})$ where $\mathcal{P}(X) = \{Y \subseteq X \mid Y \neq \{\}\}$ and for any set M, the set $M^{\mathbb{R}_+}$ stands for the set of functions from the non-negative real numbers $\mathbb{R}_+$ to M that are piecewise Lipschitz continuous, where Lipschitz continuity is extended to tuples of functions in the natural way. For functions with range different from $\mathbb{R}$ this requires that they are piecewise constant. The modularity of HyCharts results from the interface concept for components with its clear distinction between inputs and outputs. The type of Cmp defines that there is a set of possible outputs for each input, which allows us to express nondeterminism. Note that using a time synchronous model ensures that $(A \times B)^{\mathbb{R}_+} \cong A^{\mathbb{R}_+} \times B^{\mathbb{R}_+}$ (isomorphy), which we will use occasionally.

Hybrid computation. Each component which is specified by a HySChart is implemented by a *hybrid machine*, as graphically shown in Fig. 3 as a Hy-

AChart.[1]This machine consists of five parts: a time extended *discrete* (or combinational) part ($Com^{\dagger}$), an *analog* (or continuous) part (Ana), a *feedback* loop, an infinitesimal delay (Lim_z), and a projection (Out). The labels at the channels indicate their types. The feedback models the *state* of the machine. Together with Lim_z it allows the component to remember at each moment of time t the input received and the output produced "just before" t. Mathematically, Lim_z computes the limit from the left of its input at each moment of time.

The discrete part is concerned with the control of the analog part and has *no memory*. It instantaneously and nondeterministically maps the current input and the fed back state to the next state. The next state is used by the analog part to select an *activity* which specifies the continuous evolution of the machine's variables, and it is the starting state for this activity. If the discrete part passes the fed back state without modification, we say that it is *idle*. The discrete part can only select a new next state (different from the fed back state) at distinct points in time. During the intervals between these time instances it is idle and the selection of the corresponding activity is stable for that interval and defines the input/output behavior of the component during the interval. Formally, the type of Com is $(\mathcal{I} \times m \cdot \mathcal{S}) \to \mathcal{P}(m \cdot \mathcal{S})$, where $\mathcal{I}$ is the input space, and $m \cdot \mathcal{S}$ is the *program-state space*. It is defined as the m-fold disjoint union (or *disjoint sum*) of the data-state space $\mathcal{S}$. Elements of the program-state space are written as $(k, s) \in m \cdot \mathcal{S}$. Here, $k \in \{1, \ldots, m\}$ encodes the part (i.e. the addend) of the disjoint sum from which s stems. Due to the semantics definition for HySCharts k encodes the *control-state* of the machine. The data-state s can be regarded as a variable assignment. The type of Ana is $(\mathcal{I} \times m \cdot \mathcal{S})^{\mathbb{R}_+} \to \mathcal{P}(m \cdot \mathcal{S}^{\mathbb{R}_+})$. Out is a projection and provides that only parts of the state-space are visible outside.

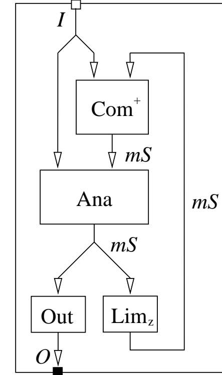

Figure 3: Hybrid machine computation model.

From syntax to semantics. From a syntactic point of view, HyACharts and HySCharts are both constructed from primitive nodes and certain macros by the application of node operators and arrow operators to build a *hierarchic graph*. By syntactic transformations (different for HyACharts and HySCharts) the macros are expanded and a plain hierarchic graph remains. According to the ideas in [GSB98b] these graphs are given a *multiplicative interpretation* for HyACharts, while the graphs for HySCharts are interpreted by an *additive interpretation* of the operators (Fig. 4). In the multiplicative interpretation the node operators correspond to independent parallel execution, sequential composition and feedback (fixed point calculation) of relations on trajectories. The definition of the operators ensures compositionality such that one can reason about components individually in order to derive properties of the composite system. The additive interpretation ensures that the operators correspond to switching, sequential composition and iteration of next state relations, i.e., of relations that have a similar type as Com. For the

[1]The textual definition is $Cmp_z = ((\mathcal{R}_2 \times \mathsf{I}) \;;\; (\mathsf{I} \times Com^{\dagger}) \;;\; Ana \;;\; \mathcal{R}_2 \;;\; (Out^{\dagger} \times Lim_z)) \uparrow_{\times}^{m \cdot \mathcal{S}}$.

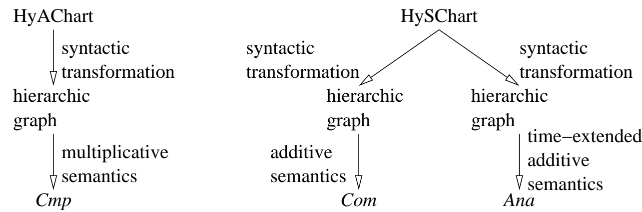

Figure 4: Principle of the semantics definition for HyCharts.

definition of the analog part a time extended version of the additive operators is defined. The time extended sum operator allows switching between different continuous input/output behaviors (or activities).

5 Mapping HyROOM into HyCharts

We outline a mapping from MaSiEd architecture diagrams, block diagrams and MaSiEd state machines to HyACharts and HySCharts, respectively (left part of Table 1). Based on HyCharts' semantics this implicitly defines the semantics of MaSiEd. The right part of the table results from the HyChart semantics definition as defined in [GSB98a, GS98].

Motivation. There are three main reasons which motivate the desire for a semantic foundation of HyROOM: validation techniques, tool integration, and (time) refinement/refactoring.

First, we are beginning to develop validation techniques for HyROOM models which are based on using ideas from model checking and logic programming for automatic test case generation (see [PL01] for related work in a purely discrete context). Since model checking and logic programming are largely based on mathematical principles, a mapping from HyROOM to such principles is mandatory.

Second, in concrete development processes the application of special purpose tools which are not integrated in the CASE tool environment provided by a single supplier is often inevitable. A mapping from one syntax to another usually is not enough, since distinct tools may interpret the same syntactic entity differently. Thus, the meaning of a notation also has to be defined and taken into account when tools are coupled. Otherwise there is an inherent danger of serious bugs resulting from slightly different interpretations.

Finally, the mapping of HyROOM to HyCharts makes a set of refactoring and refinement methods (see [St01] for the latter) accessible to HyROOM. These methods focus on the refinement of state machines (HySCharts). They make some techniques introduced in [Ru96, Sc01] amenable to HySCharts. However, their major aim is to support the transition from a specification with a *continuous time* model to specification with an underlying *discrete time* model. Due to the simplicity of the mapping from HyROOM to HyCharts, it is straightforward to transfer these methods to HyROOM. Besides design modifications which preserve vital properties, they also offer a systematic way to derive a suitable size of the

architecture diagram	$\rightarrow$	HyAChart	$\rightarrow$	$Cmp \in \mathcal{I}^{\mathbb{R}}+ \rightarrow \mathcal{P}(\mathcal{O}^{\mathbb{R}}+)$
block diagram	$\rightarrow$	HyAChart, predefined blocks	$\rightarrow$	$Cmp \in \mathcal{I}^{\mathbb{R}}+ \rightarrow \mathcal{P}(\mathcal{O}^{\mathbb{R}}+)$
state machine	$\rightarrow$	HySChart, set of initial states	$\rightarrow$	$Com \in (\mathcal{I} \times m \cdot \mathcal{S}) \rightarrow \mathcal{P}(m \cdot \mathcal{S})$ and $Ana \in (\mathcal{I} \times m \cdot \mathcal{S})^{\mathbb{R}}+ \rightarrow \mathcal{P}(m \cdot \mathcal{S}^{\mathbb{R}}+)$ resulting in $Cmp_z \in \mathcal{I}^{\mathbb{R}}+ \rightarrow \mathcal{P}(\mathcal{O}^{\mathbb{R}}+)$

Table 1: From MaSiEd via HyCharts to relations.

time steps used in the simulation by MaSiEd. Formally, if the time step is chosen appropriately, the behavior of a discrete-time simulation with this step size is a refinement of the behavior of the original system with continuous time model. Note that if a step size is chosen without the guidance offered by formal refinement methods there is no mathematically guaranteed relation to the behavior of the specified system. In any case, deeply mathematical development support become possible by means of the formal foundation of HyROOM.

MaSiEd and HyChart abstract syntax. Table 2 lists the main constituents of MaSiEd architecture diagrams, block diagrams, MaSiEd state machines, HyACharts and HySCharts. Furthermore, it indicates which concepts are mapped to each other. [SPP01] explains the mapping in greater detail, considers further MaSiEd concepts and also discusses the well-definedness of the resulting semantics, which in particular is not guaranteed for all kinds of block diagrams.

Architecture diagrams. As Table 2 indicates, the constituents of MaSiEd architecture diagrams and HyACharts are very similar. In order to obtain a HyAChart from a MaSiEd architecture diagram primitive capsules defined by block diagrams are mapped to HyAChart components; their semantics is explicitly defined by the translation of the block diagram (see below). Each capsule defined by a state machine is mapped to a primitive HyAChart component defined by the HySChart resulting from translating the state machine to a HySChart (see below). A capsule defined by a MaSiEd architecture diagram is mapped to a HyAChart component which is, in turn, defined by applying this mapping recursively to the subdiagram. For the mapping from MaSiEd connectors to HyACharts channels it is necessary to replace each bidirectional connector by two unidirectional channels in a consistent way. Apart from that some extension of HyAChart component interfaces is necessary to reflect the buffered communication associated with bindings in MaSiEd. The details are rather technical and are given in [SPP01].

Block diagrams. A mapping from block diagrams to HyACharts can be defined as follows. We limit the block diagrams we want to consider to only containing primitive blocks for arithmetic operations like addition, multiplication, division and integration, which are most important in practice.[2] Each hierarchic block is mapped to a hierarchic HyAChart component and the mapping is applied recursively to the block's defining block diagram. Block diagram channels are directly

[2]MaSiEd also allows the direct specification of blocks with C++ code. In the context of HyCharts, the semantics of such blocks would have to be provided by the designer.

MaSiEd architecture diagram	**HyAChart**
primitive capsules (with ports) as block diagrams	primitive components, semantics directly defined
primitive capsules (with ports) as state machines	primitive components as HySCharts
capsules in architecture diagrams	components as HyACharts
connectors between bidirectional ports	unidirectional channels between components
block diagram	**HyAChart**
primitive blocks, predefined	primitive compts., sem. defined explicitly
hierarchic blocks as block diagrams	components as HyAChart
unidirectional channels between blocks	unidirectional channels between components
MaSiEd state machine	**HySChart**
primitive states with transition points, entry and exit code, block diagram	primitive states with transition points, entry and exit action, activity
hierarchic states as state machine with transition points, entry and exit code, block diagram	hierarchic states as HySChart, with transition points, entry and exit action, activity
transitions between transition points, with code (guard+assignmt.)	transitions between transition points with (guarded) action
initial transition to transition point with code (assignment)	*explicit by definition of initial states*

Table 2: Main syntactic entities and their mapping.

mapped to HyAChart channels. Thus, we finally end up with primitive blocks. Each primitive block is mapped to a primitive HyAChart component whose semantics is explicitly defined according to the respective arithmetic operation. For instance, the semantics of the HyAChart components for addition + and integration *int* is defined as follows:

$$\begin{array}{lcl} + & \in & (\mathbb{R} \times \mathbb{R})^{\mathbb{R}_+} \rightarrow \mathcal{P}(\mathbb{R}^{\mathbb{R}_+}) \\ +(\iota,\kappa)(t) & = & \iota(t) + \kappa(t) \end{array} \qquad \begin{array}{lcl} int_c & \in & \mathbb{R}^{\mathbb{R}_+} \rightarrow \mathcal{P}(\mathbb{R}^{\mathbb{R}_+}) \\ int_c(\iota)(t) & = & \int_0^t \iota(u)du + c \end{array}$$

where c is a constant used as the initial value of the integrator. Since the trajectories we consider are piecewise Lipschitz continuous, the integral is guaranteed to exist for every interval where the input is Lipschitz continuous and it will result in a piecewise Lipschitz continuous output. The definitions for the other operations are similar.

Applying the semantics definition of [GS98] to the resulting HyAChart then yields a relation on input and output trajectories, parameterized with the start values of the integrator blocks of the original block diagram. This relation being the semantics of the block diagram, it needed in the translation of MaSiEd architecture diagrams and for the translation of MaSiEd state machines, which also contain block diagrams, to HySCharts.

State machines. In the mapping from MaSiEd state machines to HySCharts each primitive state with its transition points is mapped to a HySChart state

with similar transition points. We assume a semantics for the (subset of) C++ code which defines entry and exit actions and transition actions to be given as a relation on the input, the state machine's present data state and its next data state, $[\![.]\!] \in \mathcal{I} \times \mathcal{S} \times \mathcal{S}$. With this semantics the HySChart state gets entry and exit action $[\![entry]\!]$ and $[\![exit]\!]$, respectively. The block diagram associated with the state in MaSiEd is mapped to an activity of the HySChart state. This requires the semantics mapping of block diagrams as outlined above and some conversion of the resulting relation's type to suit to the type of activities. Details of this conversion as well as some further aspects concerning the triggering of transitions by block diagrams are described in [SPP01].

Every transition in the MaSiEd state machine between two transition points is mapped to a HySChart transition between the images of the transition points. The semantics of the C++ code which is specified for the transition in MaSiEd is associated with the image of the transition in the HySChart. It is defined as for entry/exit actions.

Each hierarchic state with its transition points in the MaSiEd state machine is mapped to a hierarchic state with corresponding transition points in the resulting HySChart. The hierarchic state's entry and exit action, and its associated block diagram are treated just as for primitive states (see above). Then the mapping is applied recursively to the substates and the transitions contained within the hierarchic state.

Initialization of the MaSiEd state machine is directly mapped to the semantics level of HySCharts yielding an element of the HySChart's state space. While the mapping sketched here does not cover some special state machine concepts like *history connectors* it does cover the important means of preemptive transitions. For details on initialization and these further concepts the reader is deferred to [SPP01]. The small example in the Appendix sketches a part of the result of applying the defined mapping to get the semantics of a part of the model depicted in Fig. 2.

6 Conclusion

In this paper we outlined MaSiEd, a prototype tool for modeling and simulation of hybrid systems specifically targeting at the application field of process automation. The modeling concepts, an adaption of concepts from UML, Rational Rose RealTime and ROOM [Gr00, Co01, SGW94], have been described and demonstrated along the lines of an example taken from an industrial case study. It has been argued that there is need for a formal semantics reflecting the tool's simulation semantics: A development process with refinement is desired as well as the application of formal methods to hybrid models is. Furthermore, the coupling of notations for discrete and continuous systems had to be clarified.

For the vital concepts of the notations supported by MaSiEd a mapping to the formal HyCharts notation [GSB98a] was outlined (see [SPP01] for more details; the semantics of HySCs has been defined in [GKS00]). This results in a formal semantics for the notations used in the tool. Although MaSiEd and HyCharts stem from

independent work at the authors' affiliations, the mapping itself is straightforward in large parts, since ROOM is a common root for both lines of work. (MaSiEd was developed with the aim of obtaining a practical *development tool* for test beds of PLCs, while the development of HyCharts targeted at designing a *mathematically precise formalism* for the practical modeling of hybrid systems.) However, complications arise in the mapping from block diagrams to continuous activities in HyCharts because the implementation related decisions made in MaSiEd differ in some details from the ideas underlying activities in HyCharts [SPP01].

Currently, HyROOM and MaSiEd do not include a notation of class diagrams. Their incorporation (e.g., similar to [FNW98]) is the focus of current work. We consider sound *refactoring* of (hybrid) pipe and filter architectures a crucial activity within a development process. In addition to automated test case generation, this huge domain also is the subject of ongoing work.

Acknowledgment. We would like to thank Lingxiang Xu for providing the original case study as well as Oscar Slotosch for valuable comments on a draft version of this paper.

References

[Al93] R. Alur, et al. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems I*, LNCS 736. Springer-Verlag, 1993.

[Al00] R. Alur, et al. Modular specification of hybrid systems in Charon. In *Proc. of HSCC'00*, LNCS 1790. Springer-Verlag, 2000.

[AT98] J. Albert and Tomaszunas. Komponentenbasierte Modellbildung und Echtzeitsimulation kontinuierlich-diskreter Prozesse. In *Proc. VDI/VDE GMA-Kongreß Meß- und Automatisierungstechnik*, Ludwigsburg, Germany, 1998.

[Be94] M. von der Beeck. A comparison of statecharts variants. In *Proc. of FTRTFT'94*, LNCS 863. Springer-Verlag, 1994.

[Be99] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, 1999.

[Br99] M. Broy. A Logical Basis for Modular Systems Engineering. In *Calculational System Design*, volume 173 of *NATO Science Series F.* IOS Press, 1999.

[Co01] Rational Software Corporation. Rational Rose RealTime. http://www.rational. com/products/rosert/index.jsp, 2001.

[FNW98] V. Friesen, A. Nordwig, and M. Weber. Object-oriented specification of hybrid systems using UML^{h} and ZimOO. In *Proc. 11th Int. Conf. on the Z Formal Method (ZUM)*, LNCS 1493. Springer, 1998.

[Fo99] M. Fowler. *Refactoring - Improving the Design of Existing Code.* Addison Wesley, 1999.

[GKS00] R. Grosu, I. Krüger, and T. Stauner. Hybrid Sequence Charts. In *Proc. of ISORC 2000.* IEEE, 2000.

[Gr93] R.L. Grossman, et al., editors. *Hybrid Systems I*, LNCS 736. Springer-Verlag, 1993.

[Gr00] Object Management Group. Unified Modeling Language (UML), version 1.3. `www.omg.org/technology/documents/new_formal/unified_modeling_language.htm`, 2000.

[GS98] R. Grosu and T. Stauner. Modular and visual specification of hybrid systems – an introduction to HyCharts. Technical Report TUM-I9801, Technische Universität München, 1998.

[GSB98a] R. Grosu, T. Stauner, and M. Broy. A modular visual model for hybrid systems. In *Proc. of FTRTFT'98*, LNCS 1486. Springer-Verlag, 1998.

[GSB98b] 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.

[KP92] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In J. Vytopil, editor, *Proc. of FTRTFT'92*, LNCS 571. Springer-Verlag, 1992.

[Ly96] N.A. Lynch, et al. Hybrid I/O automata. In *Hybrid Systems III*, LNCS 1066. Springer-Verlag, 1996.

[Mo99] P. J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *Hybrid Systems Computation and Control (HSCC'99)*, LNCS 1569. Springer-Verlag, 1999.

[MS00] 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.

[PL01] A. Pretschner and H. Lötzbeyer. Model Based Testing with Constraint Logic Programming: First Results and Challenges. In *Proc. 2nd Intl. Workshop on Automated Program Analysis, Testing, and Verification (WAPATV'01)*, May 2001.

[PPS00] I. Péter, A. Pretschner, and T. Stauner. Heterogeneous Development of Hybrid Systems. In *Proc. Rigorose Entwicklung software-intensiver Systeme*, pages 83–93, Berlin, 2000.

[PSS00] A. Pretschner, O. Slotosch, and T. Stauner. Developing Correct Safety Critical, Hybrid, Embedded Systems. In *Proc. of New Information Processing Techniques for Military Systems*. NATO Research, 2000.

[Ru96] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Ph.D. thesis (in German), Technische Universität München, 1996.

[Sc01] P. Scholz. Incremental design of statechart specifications. *Science of Computer Programming*, 40(1):119–145, 2001.

[SGW94] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons Ltd, Chichester, 1994.

[SPP01] T. Stauner, A. Pretschner, and I. Péter. Approaching a hybrid UML-RT: Tool support and Formalization. Technical report, TU München, 2001. To appear.

[St01] T. Stauner. *Systematic Development of Hybrid Systems*. PhD thesis, Technische Universität München, to appear 2001.

A Example: Hierarchic Graph for State OK

Fig. 5 depicts the hierarchic graph for state *OK* (and substates *Change*, *Thread* and *Wind*) of the MaSiEd state machine in Fig. 2. The graph defines the state transition relation of *OK* and results from mapping the state machine to a HySChart as described in this paper and then deriving this HySChart's discrete part by applying the graph transformations described in [GS98] (see also Fig. 4, path in the middle). The HySChart's discrete part *Com* (cf. Fig. 3) is composed from the state transition relations of the HySChart's substates just as *OK* is composed

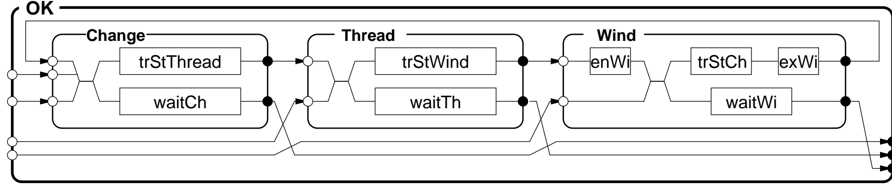

Figure 5: Hierarchic graph for state OK.

from *Change*, *Thread* and *Wind*. Hence, the given graph is a part of the definition of the HySChart's discrete part.

The equivalent textual notation for the relation depicted as a hierarchic graph above is as follows:

$$Change = {}_3{\succ\!\bullet}_1; {}_1{\bullet\!\prec}_2; waitCh + trStThread$$
$$Thread = {}_2{\succ\!\bullet}_1; {}_1{\bullet\!\prec}_2; waitTh + trStWind$$
$$Wind = \mathsf{i} + enWi; {}_2{\succ\!\bullet}_1; {}_1{\bullet\!\prec}_2; waitWi + (trStCh; exWi)$$
$$OK = (\mathsf{i} + \mathsf{i} + Change; {}^1_2\!\chi + \mathsf{i}; \mathsf{i} + \mathsf{i} + Thread; {}^1_2\!\chi + \mathsf{i}; \mathsf{i} + \mathsf{i} + Wind; {}^1_2\!\chi + \mathsf{i}) \uparrow^1_+$$

where *trStThread*, *trStWind* and *trStCh* are relations, namely the semantics of the code specified for the similarly named transitions in the MaSiEd state machine of Fig. 2. They have type $\mathcal{I} \times \mathcal{S} \times \mathcal{S}$ and are partial in $\mathcal{I} \times \mathcal{S}$. Informally, this expresses that if the trigger of the corresponding transition is false for an input and current state, the relation does not determine a next state. Relations *waitTh*, *waitCh* and *waitWi* are derived from *trStThread*, *trStWind* and *trStCh*, respectively, by some set algebra. Informally, they denote that the next state is equal to the present state if no transition is enabled. If a transition is enabled they are undefined. They also have type $\mathcal{I} \times \mathcal{S} \times \mathcal{S}$. The compound relation OK has type $\mathcal{I} \times 4 \cdot \mathcal{S} \to \wp(3 \cdot \mathcal{S})$. The further symbols in the formulas are node and arrow operators defined for hierarchic graphs in [GS98]. They formalize the kind of connections suggested by the visual representation given in Fig. 5.