Executable HybridUML and its Application to Train Control Systems*

Kirsten Berkenkötter, Stefan Bisanz, Ulrich Hannemann, and Jan Peleska

University of Bremen,
P.O. Box 330 440
28334 Bremen, Germany
{kirsten,bisanz,ulrichh,jp}@informatik.uni-bremen.de

Abstract. In this paper, the authors introduce an extension of UML for the purpose of hybrid systems modeling. The construction uses the profile mechanism of UML 2.0 which is the standard procedure for extending the Unified Modeling Language. The "intuitive semantics" of the syntactic extension is based on the semantics for hierarchic Hybrid Automata, as suggested by Alur et. al. In contrast to Alur's formalism, HybridUML allows to label transitions not only with conditions and assignments, but also with signals. Furthermore, our approach associates formal semantics by definition of a transformation from HybridUML specifications into programs of a "low-level" language which is both executable in hard real-time and semantically well-defined. When compared to approaches assigning semantics directly to the high-level constructs of a formal specification language, the transformation approach offers two main advantages: First, semantics can be more easily adapted to syntactic extensions by extending the transformation in an appropriate way. Second, all models are automatically executable, since the low-level language is.

1 Introduction

A real-time system is called *hybrid* if it processes *time-continuous* variables in addition to discrete-range parameters. The (piecewise) continuous evolution over dense time of real or complex observables occurs naturally in physical models and in the development of (embedded) control systems monitoring some continuous observables (e.g. temperature, speed) via analog sensors and setting others (e.g. voltage, thrust) using actuators.

In this paper, the authors introduce *HybridUML*, a novel specification formalism for hybrid systems, and the *Hybrid Low-Level Language Framework* HL^3 for generating programs to be executed in hard real-time on cluster hardware architectures.

* The work presented in this article has been investigated by the authors in the context of the HYBRIS (Efficient Specification of Hybrid Systems) project supported by the Deutsche Forschungsgemeinschaft DFG as part of the priority programme on *Software Specification – Integration of Software Specification Techniques for Applications in Engineering.*

H. Ehrig et al. (Eds.): INT 2004, LNCS 3147, pp. 145-173, 2004.
© Springer-Verlag Berlin Heidelberg 2004

As suggested by its name, HybridUML is based on the syntax of the Unified Modeling Language UML 2.0 [OMG03a,OMG03b]. Since core UML does not support the specification of time-continuous behavior, a language extension is required. To this end, the authors introduce a new *profile*, that is, a definition of new UML constructs introduced by means of UML stereotypes applied to existing language elements. In particular, HybridUML extends the UML variant of Statecharts by augmenting state descriptions using *invariants* and *flows* or *algebraic conditions*. The latter ones describe time-continuous variable evolutions taking place while the system resides in the respective state. Transitions may be triggered by conditions and signals (i.e. atomic events) and lead to actions consisting of variable assignments and signal generations. Following the suggestions of existing formalisms [Hen96,AGLS01], transition execution is conceptually performed in zero-time, but with interleaving semantics. Parallel components process signals in a synchronous, but non-blocking way: A signal s generated by some transition is available in multicast-fashion to other agents for their next computation step. Time passes during the phases where agents reside in a given state, and time-continuous variables change according to the flow/algebraic conditions applicable in the current agent states.

The HL^3 framework developed by the authors consists of a re-usable hard real-time runtime environment R and a design pattern P for compilation targets of arbitrary hybrid specifications. Given a high-level formalism H – such as HybridUML – for the description of hybrid systems, transformations Φ_H from high-level specifications S into instances $\Phi_H(S)$ of the HL^3 pattern P can be developed. For $(\Phi_H(S), R)$, a formal semantics $\mathcal{S}(\Phi_H(S), R)$ is defined so that the transformation both provides a semantic definition of S and an executable program whose behavior will be consistent with $\mathcal{S}(\Phi_H(S), R)$. Similar to machine code, HL^3 should not be used for manual programming, but as a target language for automated transformations. In contrast to machine code, the real-time semantics of HL^3 programs can be determined in a direct way, thereby assigning formal meaning to the high-level specification used as the transformation source. This is achieved by using a very limited range of instructions for multi-threading, timing control, and consistent handling of global state in presence of concurrency.

Though today numerous formalisms and verification approaches are available for hybrid systems (see references in the related-work section below), their application in an industrial "real-world"-context is still rare. According to our analysis, two main causes are responsible for this situation:

- The syntax developed for hybrid formalisms within research communities was too specialized and not supported by conventional software engineering tools available to practitioners.
- While the underlying theories supported formal verification by theorem proving or model checking, they did not support the development of optimized code for embedded control systems.

With respect to the first cause we suggest to augment existing well-accepted formalisms of software engineering by new specification constructs describing

time-continuous behavior. From today's point of view, the Unified Modeling Language UML 2.0 is the best candidate for such an approach: It is currently the most widely known software-engineering formalism supported by a variety of tools. Furthermore, language extension is an inherent feature of UML, therefore well-constructed UML tools should support this extension as well.

The second cause is related to both practical and theoretical considerations: From a practitioner's point of view, the effort invested into formal specification and verification – which will certainly be considerably higher than the effort spent on elaborating informal conventional specifications – is only justified if the specifications can be easily transformed into executable systems. For example, we do not expect that the amount of time required for developing executable code by step-wise refinement will ever be widely accepted among project leaders and developers of embedded systems.

From a theoretic point of view, the problem is even more subtle: If a transformation into executable code is available, how can the consistency between high-level specification semantics and execution behavior of the low-level implementation using conventional programming languages and operating systems be ensured? A practical consequence of this problem consists in the fact that the simulation facilities provided by many case tools never declare which formal high-level semantics has been used as a reference for the encoded simulation behavior.

In "classical" UML [RJB99,OMG03a,OMG03b], the definition of a universal formal semantics has been deliberately avoided. Instead, the various language constructs are only associated with a general informal meaning so that their purpose in various modeling situations becomes clear. In [RJB99, pp. 105] this approach is motivated by the fact that the semantic interpretation of specification constructs depends on the specific project context, and precise behavior is only obtained by transformation into the target programming language. While this avoids the obligation to prove consistency between executable system and high-level specification semantics, it still poses the problem that in general, it will be infeasible to capture the potential behavior of software written in Java, C/C++, or Ada, when executed in a specific target environment.

Our suggestion to overcome this problem is to restrict the infinite variety of possible compilation targets for hybrid specifications according to the HL^3 framework introduced below: First, the framework fixes a specific hard real-time runtime environment which avoids uncertainties introduced by using arbitrary operating systems. Second, all specifications written in a given hybrid high-level formalism have to be compiled using a transformation function which generates instances of abstract classes pre-defined by the framework. As a consequence, the variable compilation targets depending on formalism and specification are restricted with respect to software architecture and interfaces to the runtime environment. Therefore the behavioral semantics of the executable target can be given more easily than for an unrestricted compilation into a programming language. If the high-level formalisms have been introduced informally, the transformation defines the semantics as well. If, however, the transformation has only

been created in order to translate specifications with given high-level semantics into executable code, the consistency between abstract specification behavior and executable compilation target still has to be verified. Due to the restrictive structure of compilation targets and runtime environment, this proof obligation is at least easier to discharge within the HL^3 framework than for arbitrary transformations designed in an intuitive way.

Before presenting a more formal definition, the "look-and-feel" of the new HybridUML profile is illustrated in Section 2 by means of a train control systems case study defined by the DFG priority programme *Software Specification – Integration of Software Specification Techniques for Applications in Engineering* [DFG]. The UML 2.0 profile defining HybridUML in a systematic way is described in Section 3. For illustration purposes, these definitions refer to the case study introduced before. Conforming to the general UML approach, the profile defines some basic semantic features together with the syntax, but is still quite far from a complete formal description of behavioral properties. To achieve this, we first introduce the HL^3 framework in Section 4. This is used in Section 5 to specify the full HybridUML semantics by providing a transformation into HL^3. Apart from describing the specification capabilities offered by the HybridUML profile, this paper focuses on the transformation concept and the semantic model of the HL^3 framework. Other areas of interest – such as hard real-time simulation, automated test data generation against HybridUML specifications, and performance measurements of HL^3 implementations on multi-CPU computer clusters – are briefly discussed in the conclusion (Section 6), with references to current work in progress.

Due to the usual space limitations, the HybridUML profile definition, the HL^3 semantics, and its performance as a time-triggered hard real-time runtime environment on cluster architectures cannot be exhaustively described in this paper. We refer to [BBHP04] for a detailed description of HybridUML profile and [BBH$^+$04] for semantics, implementation, and real-time measurements of HL^3.

Hybrid systems have been studied extensively in various research communities since the early nineties. The definition and investigation of the Duration Calculus (see [ZRH93,RRS03] and further references given there) provided fundamental contributions to understanding Hybrid Systems. The introduction of Hybrid Automata [Hen96] demonstrated the feasibility of verification by model checking for hybrid specifications. The applicability of hybrid automata to large-scale systems was improved by the introduction of hierarchical hybrid specifications [AGLS01]. Alternative hierarchical approaches closer to the Statecharts formalism have been described in [KMP00] (together with a proof theory) and [BBB$^+$99] (verification by model checking).

We mention GIOTTO [HHK03] as today's most prominent example of a hard real-time language with well-defined semantics. Similar to our HL^3 framework, GIOTTO follows the time-triggered systems paradigm described in [Kop97]. The time-triggered approach is particularly well-suited for real-time programs discretising time-continuous evolutions, since it guarantees bounded timing jitter

for periodic schedules. In contrast to this, other approaches to hard real-time focus on the fast response to external interrupts, see [RTAI03,Lab04] for popular real-time variants of the Linux operating system.

2 HybridUML by Example: Radio-Based Train Control

In this section, HybridUML is introduced and illustrated by means of an application example – the specification of a radio-based train control system. This is one of two case studies within the scope of the DFG priority programme Software Specification [DFG].

HybridUML as Profile for UML 2.0 One of the mostly critized points of UML 1.4 is the lack of formal specification. Especially the real time community needs this for building safe systems. As this fact has not changed with UML 2.0, another solution must be found for using UML in the real-time domain.

To overcome this deficiency, we propose HybridUML as a UML 2.0 profile. UML 2.0 offers profiles as a powerful extension mechanism for tailoring UML to specific working areas. Based on a metamodel like the Meta Object Facility (MOF) or usually UML itself, a profile specifies new model elements called stereotypes. Each stereotype is dependent on exactly one element of the corresponding metamodel (see Fig. 5 for an example). These stereotypes customize the used metamodel in different ways: introducing a new terminology, e.g. for Enterprise Java Beans, introducing new syntax, either for elements without syntax or new symbols for elements with syntax, introducing new semantics and constraints, or adding further information like transformation rules from model to code. A set of stereotypes forms a profile.

A profile can be applied by a model or a package in a model. All stereotypes can be used as modeling elements. As every stereotype extends an already known element, the model is still a valid UML model if the profile is taken away. Profile application is visualized by a dependency with the keywort ≪*apply*≫ attached. The profile itself is a package and therefore depicted like this with the keyword ≪*profile*≫ above its name. The HybridUML profile thus takes a subset of UML, modifies it according to the requirements on a specification formalism for hierarchical hybrid systems, and associates it with a precise semantics. The most important constraint on applying the HybridUML profile is using only the model elements specified in it.

Radio-Based Train Control An important part of the specification of the train control system is the coordination of train and railroad crossings in order to ensure that whenever the train crosses the road, the crossing is safe (i.e. it is locked for cars, pedestrians etc.). A special feature is the absence of signals and train monitoring equipment on the track, i.e. train and crossings always guarantee a safe and consistent state of the complete system on behalf of state requests and notifications. Particularly, the train controller continuously (re-)calculates velocity-dependent locations on the track at which requests must be sent, or at

which the brake has to be activated. There are related investigations concerning the "generalized railroad crossing" applying a more abstract model of the system, e.g. [HL94].

In this case study, a single train is considered that moves on a single track without switches. There are *velocity target points* on the track with dedicated velocities that the train must not exceed. There are two kinds of velocity target points: (1) *Conditional* velocity target points are assigned to railroad crossings and have a target velocity $v = 0$. If the crossing is not safe, the velocity target point is active and the train has to stop there. (2) The route atlas contains a (piecewise constant) static velocity profile that defines a maximum allowed velocity for each location on the track. Each location for which this velocity gets lower implies a *fixed* velocity target point, denoting a restrictive velocity change. These points are always active.

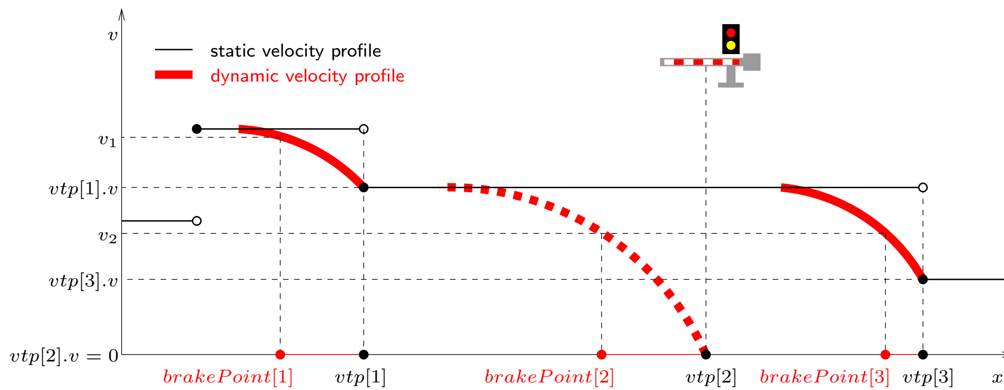

Fig. 1. Dynamic velocity profile defined by velocity target points on the track

Figure 1 is an abstract view on a track example with fixed $(vtp[1], vtp[3])$ and conditional $(vtp[2])$ velocity target points. In consideration of the train's maximum deceleration, the static velocity profile and the velocity target points define the *dynamic* velocity profile, resulting in *brake points* – the locations on the track where the train must start braking in order to reach the target velocity at the respective target point. The brake points depend on the train's current velocity; in the diagram, example values for current velocities v_1 and v_2 are given, whereas v_1 leads to brake point $brakePoint[1]$ and v_2 implies $brakePoint[2]$ and $brakePoint[3]$, respectively. A safety-critical aspect of the train control system is to ensure that the speed limit of target velocity points is never broken. The remainder of this section focuses thereon.

Architectural Structure The main building block for modeling architectural structure within HybridUML is called *agent* in conformance with related work. Agents are concurrently operating entities and can be combined by parallel com-

position, or grouped together enclosing them with a hiding operator. For precise interface descriptions we distinguish local and global variables and signals. HybridUML allows communication between concurrent agents via shared variables as well as via message passing to model multicasting of signals.

Agent TrainController consists of several (parallel) agents as defined in its structure diagram (Fig. 2), including their interdependencies. It calculates and provides the required acceleration a of the train. Therefore the train's location x and the user-requested acceleration a_{user} from the locomotive driver are read from the environment. These are time-continuously changing variables. Further, several locally defined constants affect the computation of a, like the route atlas ra which is defined as a data structure. It contains the velocity target points vtp that consist of a location x, a target velocity v, and its type (fixed or conditional). Finally, radio messages are received that provide status information about the railroad crossings.

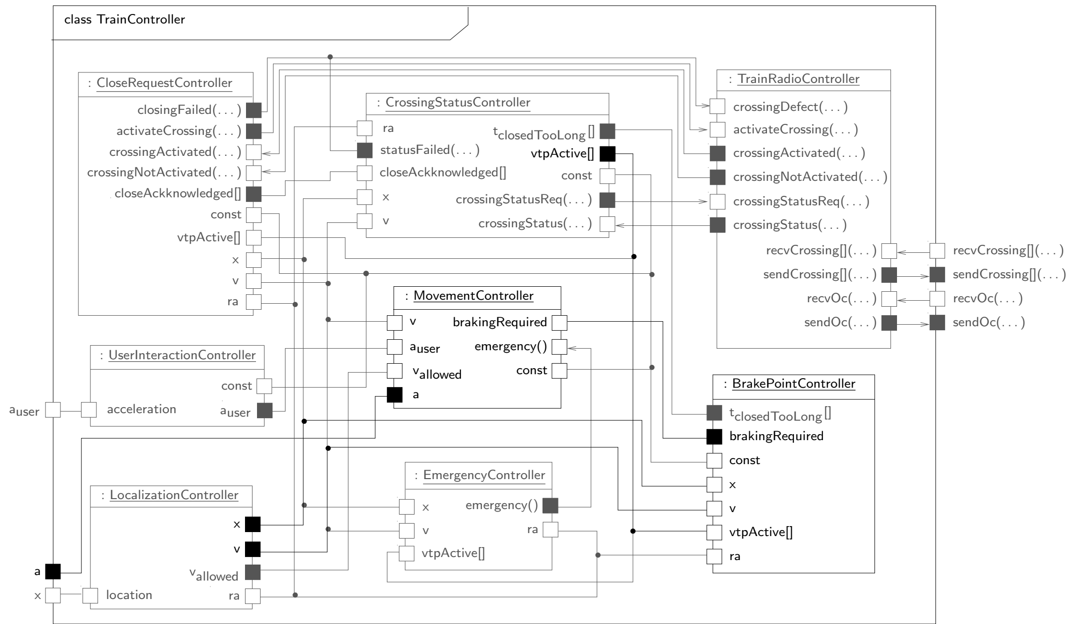

Fig. 2. Structure diagram of agent TrainController.

The highlighted basic agents BrakePointController and MovementController are discussed in the following.

Agent BrakePointController provides the boolean variable brakingRequired that denotes that the train has to brake because of any velocity target point. This is determined by the train's location x, its current speed v, and the set of

currently active velocity target points (which is defined by vtpActive[] consisting of a boolean variable for each velocity target point).

Behavioral Specification The behavior of a basic agent is given by an associated hierachical state machine. As typical for hybrid systems, there are basically two ways of acting: either some discrete transition is taken or time passes and the continuous variables change over time according to their specified constraints.

The behavior definition (shown in Fig. 3) is based on the continuous (re-)calculation of the set of brake points brakePoint[i] for all velocity target points:

(1) **algeBrakePoint** $\equiv$
$\forall i \in \{1..VTP_COUNT\} \bullet brakePoint[i] = ra.vtp[i].x - \frac{ra.vtp[i].v^2 - v^2}{2 \cdot const.a_{min}}$

The variable brakingRequired is set in a discrete fashion, dependent on condition:

(2) **condBrakingRequired** $\equiv$
$\exists i \in \{1..VTP_COUNT\} \bullet$
$vtpActive[i] \wedge brakePoint[i] \leq x \wedge ra.vtp[i].v < v \wedge ra.vtp[i].x > x$

It denotes the situations that require braking because at least one brake point of an active velocity target point is reached by the train while its speed is too high. Note that only velocity target points in front of the train are considered, because particularly the opening of a crossing behind the train shall not affect it.

Thus the mode BrakingRequired is active for exactly these situations, whereas the mode BrakingNotRequired is complementary. The transitions in combination with the invariants model the mandatory mode changes according to the condition such that variable brakingRequired is always up-to-date.

The responsibility of agent MovementController (Fig. 4) is to determine the required acceleration a. It constrains the user-requested acceleration $\mathsf{a}_{\mathsf{user}}$ on behalf of brakingRequired, the current velocity v, the currently allowed velocity $\mathsf{v}_{\mathsf{allowed}}$, and a special signal emergency. It is modeled strictly hierarchically, initially in mode normal. Submode userControlled maps the user-requested acceleration to a, whereas submode enforcedBraking forces the train to brake. Similarly to BrakingRequired, the active submode directly depends on a condition:

(3) **condBraking** $\equiv$
$brakingRequired \vee v > v_{allowed}$

In case of enforced braking, the submode braking defines a to be the maximum deceleration until the train stops. Otherwise, there is a distinction between unrestricted and restricted appliance of $\mathsf{a}_{\mathsf{user}}$: The mode restricted guarantees that the minimum (0) and maximum ($\mathsf{v}_{\mathsf{allowed}}$) velocities are not violated, else unrestricted maintains $\mathsf{a} = \mathsf{a}_{\mathsf{user}}$. Again, a condition controls this:

(4) **condUnrestricted** $\equiv$
$(v < v_{allowed} \vee a_{user} \leq 0) \wedge (v > 0 \vee a_{user} \geq 0)$

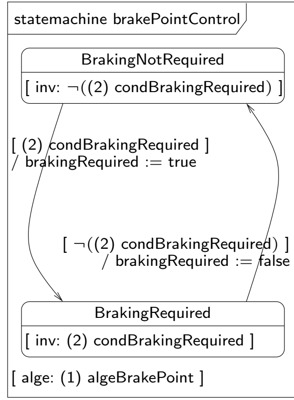

Fig. 3. Behavior of agent BrakePointController.

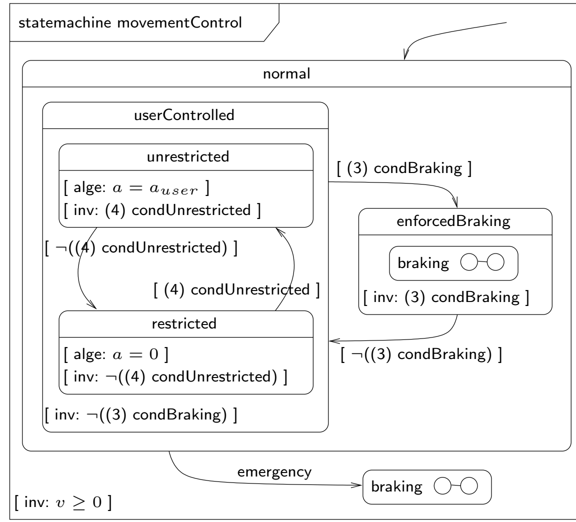

Fig. 4. Behavior of agent MovementController.

Finally, mode braking is re-used in movementControl – if the signal emergency (that is caused by violating a velocity target point) is received, the train is definitely stopped.

3 The HybridUML Profile

In this section we give a brief overview on the most relevant elements of the HybridUML profile, their relation to existing UML constructs, and their intuitive semantics. For a detailed language description we refer the reader to [BBHP04].

Types and Expressions HybridUML uses typed variables. As UML provides only Integer, String, and Boolean as basic types, we have to extend explicitly PrimitiveType to get the datatype Real. In this way, we can use real-valued variables within the profile. For better separaton of concerns, we also need analog real numbers as extension which can be changed continuously according to flow conditions in Modes while variables of type Real can only be changed discretely by transitions. AnalogReal thus is a specialization of Real. For better readability in large applications, we introduce StructuredDataType as an instance of DataType to define a structure. All StructuredDataTypes of a model are implicitly collected in a package which is imported by all diagrams of this model.

For describing the valuation of AnalogReal variables, specific expressions are needed, i.e. differential expressions and algebraic expressions. Invariant expressions are needed to define state invariants. RTExpression is an instance of

Expression (see Fig. 5) which defines mathematical and logical terms that may be dependent on time. RTExpression is an abstract metaclass that cannot be instantiated.

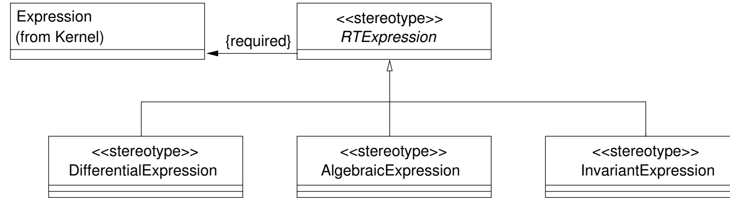

Fig. 5. Stereotypes for RTExpressions

The real-time expression is given as a string, just as in Expression. Furthermore, the expression must be mathematically or logically evaluable. The notation and semantics are given by the concrete subtypes, i.e., specializations of RTExpression, which are: *AlgebraicExpression*, to describe algebraic terms dependent on time, *DifferentialExpression*, to describe differential terms dependent on time, and *InvariantExpression*, to describe logical terms used for modeling invariants a variable must fulfill.

Constraints To describe the restrictions on the valuation of analog variables in modes we introduce RTConstraints. As an instance of Constraint, RTConstraint is a UML constraint which is restricted in order to describe an RTExpression. DifferentialExpressions and AlgebraicExpressions can be attached to AnalogReal variables, InvariantExpressions can be attached to all variable types. RTConstraint is visualized in the same way as UML 2.0 constraints, i.e. an RTExpression term given in curly brackets. In Modes, brackets are used.

Clocks and Timers We do not use the UML 2.0 time model as it has no formal semantics and as it is not powerful enough for our purposes. A Clock is modeled by a variable of type AnalogReal that uses a DifferentialEquation for modeling the flow of time. Therefore we inherit from AnalogReal to get a clock. The flow of time is specified as a differential equation: Let t be the value of a Clock instance. Then the following expression always holds: $\dot{t} = 1$. As the differential equation is explicitly given, it is not added as a constraint following the variable. Similarly we have timers which are set with a value and count downwards, consequently they are specified by the differential equation $\dot{t} = -1$.

Variables and Signals Variables in HybridUML can be shared between agents for communication purposes. They are visualized in the same way as UML 2.0 Ports, which are linked by connectors. The shared variable model requires connected interfaces to hold the same value. VariablePorts are depicted as a rectangle on the boundary of the owning classifier. Instead of visualizing the attached interfaces in lollipop-notation, a required interface (corresponding to read-only access) is a white filled rectangle and a provided (corresponding to read/write

access) interface is a black filled rectangle. In class diagrams, only the variable owned by the VariableInterface of the port will be shown.

RTSignals are introduced as a different means for communication between Agents, as pure communication via shared variables can be managed when modeling small systems, but tends to be cumbersome for larger systems. RTSignal is an instance of Signal (from Common Behaviors) which defines an asynchronous message. An RTSignal is depicted in composite structure diagrams in correspondence with SignalPorts and SignalInterfaces similar to shared variables.

Using these elements of UML allows to represent the communication structure between agents in a composite structure diagram (see Fig. 2) as far as their shared variables and signals are concerned. In statemachine diagrams, RTSignals are used in combination with SignalEvents and ModeTransitions. SignalEvents carry RTSignals. They are used as triggers in Modes. Nevertheless, we prefer the term event as this is usual for state machine models.

Agents Agents are stereotypes of classes and consist of VariablePorts, SignalPorts, private variables, Modes, initial states, and parameters. Initial states are specified in Agent instances just as concrete values for parameters. Modes are class variables and cannot be changed by Agent instances. Parameters are used for better scalability. They specify constants that can be used in invariants and other expressions used in the Agent instance and its Mode(s).

An Agent instance can own an internal structure which may consist of Agent instances itself. Agents communicate by shared variables (represented by VariablePorts and VariableConnectors) and signals (as modelled by SignalPorts and SignalConnectors).
In the HybridUML profile we distinguish *basic* Agents which are not nested and own a single top-level Mode, and *composite* Agents which are composed from subagents and have many top-level Modes. Clocks are global for all parts of an Agent. We do not model them as VariablePorts as this would be obfuscating. Parameters are constant global variables for usage in constraints of all kind. The top-level Modes define the behavior of the system. The semantics of a basic Agent are defined by the (trace) behavior of its top-level Modes, constructed from the respective relations describing the continuous behavior and the discrete transitions of a Mode (see below).

The standard operations on concurrent components like the composition of two Agents $A_1 \| A_2$, application of a hiding operator, and renaming of the variables of an AgentInstance, can be reflected in the representation of an Agent's internal structure in a composite strucure diagram. An execution of an Agent A follows a trajectory, which starts in some initial state and is a sequence of flows, i.e. continuous changes to the analog variables, interleaved with discrete steps of agents. While continuous steps are performed simultaneously by all Agents, discrete steps are performed by one Agent at a time, possibly changing variables or taking part in communication via events.

Agents are depicted like UML classes with internal structure. In a class diagram, the internal structure is visualized as aggregated classes. The parameter list of each Agent is given behind its name in parentheses in the first compart-

ment of the class symbol. VariablePorts and their included VariableInterfaces and variables as well as SignalPorts and their included SignalInterfaces and signals are given as attributes in the second compartment of the class symbol. In the class diagram, for variables only the name and type are shown, for signals the name and parameters are given. Optionally, in the third compartment of the class the Mode of the Agent is given. This is the name of the Mode followed by concrete parameters listed inside parentheses. A parameter of a Mode may also be a parameter of the Agent, i.e. the concrete value is given in an Agent instance.

The internal structure of composite Agents is shown in a composite structure diagram (see as example Fig. 2). The name of the Agent is given in the upper left corner with the keyword *class* before it. After that, the concrete parameters of the composite Agent follow. Here read/write access of global variables is shown as ports with required and provided interfaces. The same holds for sending and receiving signals.

Agent instances are visualized as objects in composite structure diagrams. Behind the objects' name and type the concrete parameters are given in parentheses in the first compartment of the object symbol. In the second compartment, optionally the respective initState is given as a constraint, i.e. in curly brackets. The Mode of the Agent is given in a statechart diagram. The name of the Mode with the keywort *statemachine* before it is given in the upper left corner of the diagram.

Modes Sets of states of a basic agent are described by *Modes* which may contain submodes themselves and transitions. In our profile, Mode is an instance of StateMachine describing an Agent's behavior. As Modes may have flow conditions, they are (hierarchical) hybrid state machines. Each Mode contains exactly one region, i.e. there is no parallel behavior inside a Mode. It is entered and left by control points, which are partitioned into entry and exit points. Top-level Modes are connected to an Agent. They use the global variables and signals defined in this Agent. Modes can have parameters for better scalability. Within Modes, the time-continuous behavior is defined by differential equations and algebraic constraints and limited by invariants. When a mode is executing a continuous step, the hierarchical state machine as a whole is acting, i.e. modes on all levels, from the top-level mode to the leaf modes, have to coordinate for this. Part of this coordination is that any possible valuation of analog variables has to comply to the constraints attached to all active modes on the various levels.

Discrete steps are described by transitions between modes where taking a transition does not take time. Transitions consist of a condition part and an action part. As condition, we can have a boolean expression, i.e. firing that transition depends on the state, or a signal trigger, i.e., an event based invocation, or both. As action of a transition we allow instantaneous operations on variables and sending of a signal. When a transition is taken, discrete variables may be updated or signals can be sent and received. Preemption and interrupts are

modeled by using group transitions, i.e. exiting a higher level Mode via some distinguished exit point.

An execution of a Mode consists of a sequence of steps which can be chosen out of three different types: a continuous step according to the respective constraints, a discrete step according to the condition and action of a transition within the mode, and an environment step that can change all but the local variables of that mode. The last variant represents an activity of a different Mode on the same hierarchical level. For a top-level mode no environment steps in this sense are possible, as it is the solitary Mode of that level.

Modes are visualized the same way as UML 2.0 StateMachines (see Fig. 4). Parameters are given behind the name of the Mode in parentheses. The invariant is marked *inv*, the flow conditions with *flow*, and algebraic expressions are marked with *alge*. As these are constraints, they are given in brackets.

Transitions In order to provide a clear-cut interface of Modes while avoiding inter-level transitions we use control points for Modes as sources and targets of transitons. Entry points are depicted as small circles on the border of a Mode with an optional name attached to it while exit points are depicted as small solid black-filled circles. Every Mode has one default entry point and one default exit point which are not depicted explicitly. We employ ModePseudostate as an instance of Pseudostate to denote control points.

ModeTransition is an instance of Transition depicted by an arrow with open arrowhead. ModeTransitions are taken according to their condition part, i.e., their guard constraints or a triggering SignalEvent. The guard constraint is given in brackets followed possibly by the SignalEvent. The ModeTransitionActivity is separated from the guard by a slash. In UML 2.0, Transitions can only have Activities as effect. As ModeTransitionActivity thus is an instance of Activity, it can be an updateActivity which updates variables according to some instruction, or it can be a sendActivity which emits a SignalEvent.

4 HL^3 – the Hybrid Low-Level Language Framework

As indicated in Section 1, the *Hybrid Low-Level Language Framework* HL^3 is a generic compilation target for hybrid high-level formalisms H. It has been designed to support the transformation Φ_H of specifications S written in H into executable code $(\Phi_H(S), R)$, thereby assigning a formal semantics $\mathcal{S}(\Phi_H(S), R)$ to the compilation target. The generated HL^3 program is suitable for hard real-time execution, to be used either for developing embedded applications or for their automated test in hardware-in-the-loop configurations. The concepts described here have been implemented on multi-CPU computers where CPUs can be reserved exclusively for the HL^3 runtime environment. In order to support executability on specific target hardware, high-level specifications S consist of three parts: (1) The *behavioral specification* S_1, written, for example, in HybridUML, (2) the *architectural specification* S_2 describing the available cluster nodes, CPUs, hardware interfaces and the mapping from S_1-objects to concrete

hardware, (3) the *physical constraints specification* S_3 describing the required frequencies for the discretization of time-continuous evolutions, writing to/reading from hardware interfaces, as well as the required precision for discrete time-dependent steps.

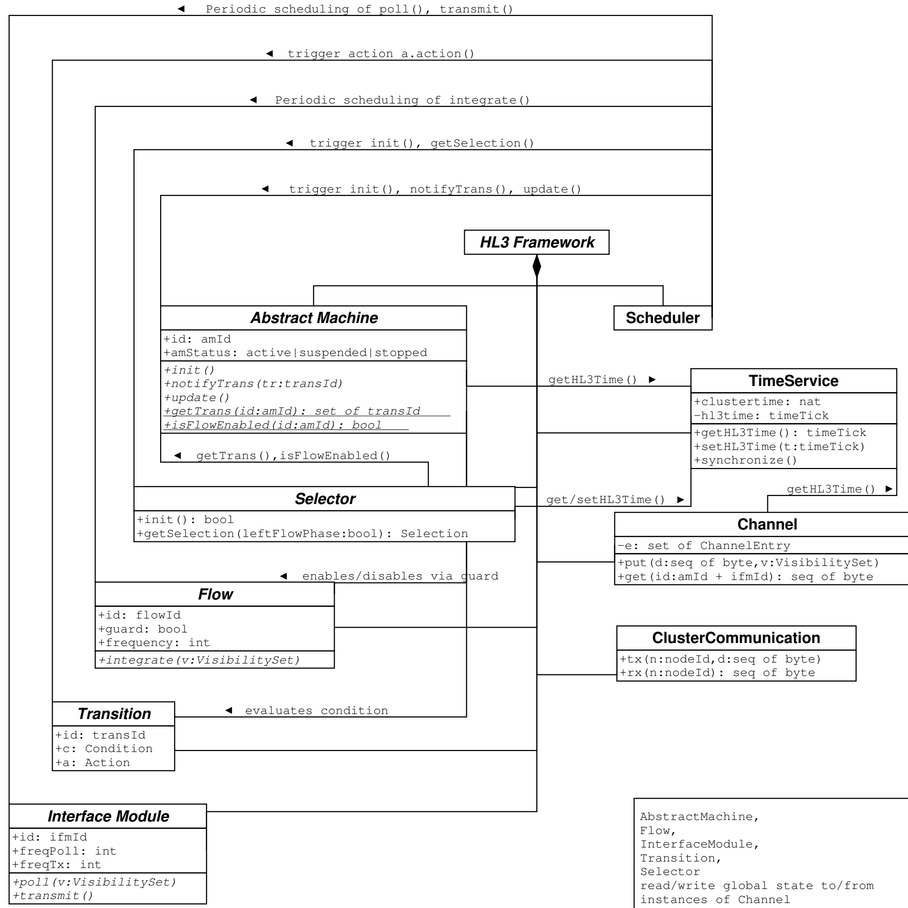

Fig. 6. Design pattern for the HL^3 run-time environment.

The framework is sketched in Fig. 6, with additional definitions of basic types shown in Fig. 7. Its underlying idea is to provide a re-usable hard real-time processing infrastructure – the *runtime environment* R – and a design pattern P for the formalism- and specification-dependent components to be executed within the runtime environment.

The HL^3 runtime environment R consists of a user space Scheduler running on reserved CPUs without interruptions from the underlying operating system

(Linux with a kernel-extension developed by the authors' research group). In addition, TimeService and a communication service (Channel and ClusterCommunication) provide the mechanisms to ensure consistent data views within the cluster configuration. The Scheduler enforces *time-triggered* real-time system behavior [Kop97]: All activities to be performed by the components of a HL^3 instance are scheduled at pre-determined points in time which are multiples of a fixed time unit.

The design pattern P consists of the abstract classes AbstractMachine, Selector, Transition, Flow, Interface Module, and their relations with each other and with the runtime environment R. Pattern P facilitates the development of the transformation Φ_H by defining the abstract interfaces and relationships we regard as essential for creating the full compilation target. Instances of AbstractMachine are used to implement the local behavior of sequential components specified in the high-level formalism. Each new high-level specification S gives rise to a new set $\alpha_H(S)$ of abstract machines, to be generated by a transformation α_H. The Selector enforces global behavioral constraints on the concurrent systems, such as the synchronous execution of transitions. Since these constraints depend on the formalism H, but not on the concrete specifications S, Selector has to be instantiated just once for each new high-level formalism H. As we are dealing with hybrid systems, sequential components may run through discrete and time-continuous processing phases. Discrete steps are represented in HL^3 by instances of Transition, time-continuous ones by instances of Flow.

For handling application-specific hardware interfaces, another abstract specification is given: Instances of Interface Module create a hardware abstraction layer, hiding driver-specific details and the location of hardware interfaces within the cluster from scheduler, flows, and transitions.

As a result, the complete transformation Φ_H from H into executable HL^3 instances can always be structured as

$$\Phi_H(S) = (\alpha_H(S), \tau_H(S), \phi_H(S), \iota_H(S), \mathsf{Selector}_H)$$

where α_H, τ_H, ϕ_H, and ι_H generate collections of abstract machines, transitions, flows, and interface modules, respectively.

In the paragraphs below, additional details of HL^3 are presented. For a complete description, the reader is referred to [BBH$^+$04].

TimeService Executable HL^3 systems are clusters connected by high-speed local area networks. As a consequence, relativistic effects between cluster nodes may be neglected, so we can assume the existence of global physical time, denoted by $@t \in \mathbb{R}_+$ with physical unit [sec]. The HL^3 TimeService provides a *cluster time* value *clustertime* $\in \mathbb{N}$ (corresponding to the notion of *global time* in [Kop97]) which is related to physical time according to

$$@t = \gamma \cdot \mathit{clustertime} + \omega + \pi, \text{ where } \pi \in [-\gamma - \Pi, \gamma + \Pi]$$

Constant $\gamma \in \mathbb{Q}$ is the cluster time *granularity* with physical unit [sec], ω is a constant to be added because *clustertime* begins at 0 on cluster startup, and Π is the precision of the cluster time, taking into account physical clock drift

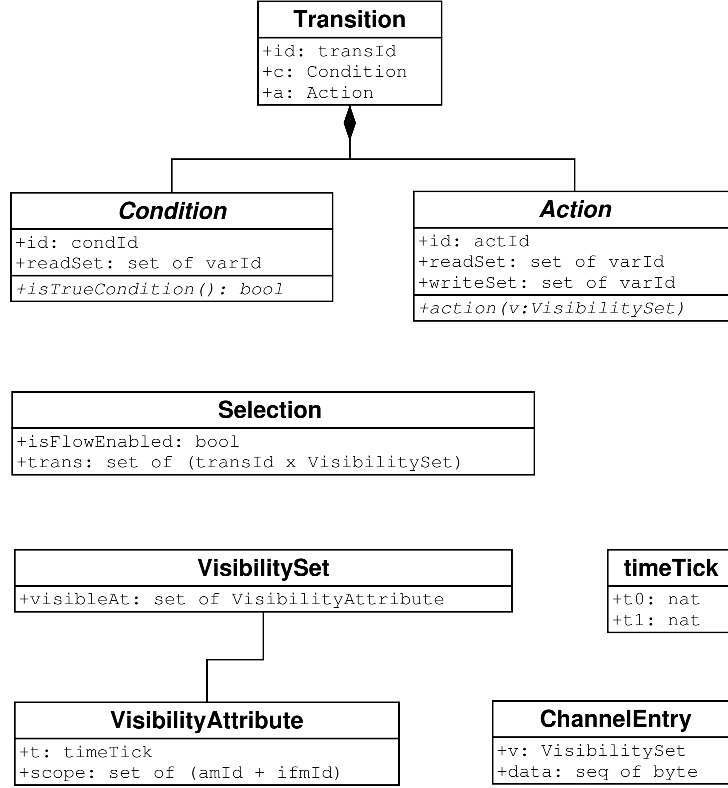

Fig. 7. Basic types referenced in the HL^3 framework.

and jitter between clocks at local cluster nodes. Typically, $10^{-9} \leq \gamma \leq 10^{-6}$ and precision values $\Pi \leq 10\mu$sec are feasible, using combinations of software and hardware clock synchronization mechanisms.

Logical HL^3-time can be obtained by all HL^3 programs as a pair $t_0.t_1 \in timeTick$ with type $timeTick = \mathbb{N} \times \mathbb{N}$, and the natural ordering $a_0.a_1 \leq b_0.b_1$ iff $a_0 < b_0 \vee a_0 = b_0 \wedge a_1 \leq b_1$. Component t_0 represents the time tick as visible to the time-dependent conditions evaluated in abstract machines. It is always ensured that $t_0 \leq clustertime$, but – depending on the high-level formalism to be encoded in HL^3 – t_0 may be kept constant for several ticks of $clustertime$, in order to simulate the execution of transitions in zero time. The second component t_1 of the logical HL^3-time is used to distinguish causally related events which occur during the same time tick t_0. Component t_1 is reset when t_0 is incremented and increased between different transition scheduling phases.

Within the HL^3 framework, the Selector component is responsible for maintaining the desired relation between logical HL^3-time and $clustertime$ via the `setHL3time()` operation provided by TimeService.

Channels One of the main objectives of the HL^3 infrastructure is to provide a consistent view on global model data. This is motivated mainly by three requirements: (1) To our knowledge, all existing high-level formalisms are based on an *atomic* transition concept: For processing transition $[C(x_1, \ldots, x_n)]e/a(x_1, \ldots, x_n)$, it is assumed that the valuations of all variables $(x_1, \ldots, x_n)$ referenced in condition or action expressions do not change while the expressions are processed. Even in formalisms which do not postulate zero-time for transition execution it is always assumed that the calculation is performed atomically – meaning in zero-time – but time passes before the earliest possible point in time when the next transition can be fired. (2) The discretization of time-continuous flows requires that all flows synchronously performing a Δt integration step view the same pre-state of all observables, even if only one CPU is available for processing quasi parallel integration steps. In particular, state changes performed by one flow must not become visible to other flows referencing the related variables before the actual integration step has been completed. (3) As the HL^3 framework supports cluster hardware architectures, a consistent view of data at all cluster nodes is mandatory.

To implement these requirements, HL^3 uses abstract data types called Channels. Channels are data containers providing two operations (see Fig. 6): The `put(d:seq of byte,v:VisibilitySet)` method stores data items d in the channel c, together with a VisibilitySet v. The elements of v are VisibilityAttributes $a = (t, scope)$, where $t = t_0.t_1$ is a logical HL^3 time tick and $scope$ a set of abstract machine or interface module identifications. The interpretation of a is as follows: When retrieving the data from a channel using the `d = c.get(id)` command, the returned value d satisfies the following constraints:

$$\exists x \in c.e, a \in x.v.visibleAt \bullet (x.data = d \wedge id \in a.scope \wedge a.t \leq \texttt{getHL3Time()}) \wedge$$
$$(\forall y \in c.e, b \in y.v.visibleAt \bullet (id \in b.scope \wedge b.t \leq \texttt{getHL3Time()} \Rightarrow b.t \leq a.t))$$

Intuitively speaking, the identification id of the caller is a member of a scope attribute set associated with d. Among all data items contained in the channel such that id is within their scope, d is the most recent entry associated with a visibility time attribute which is less or equal to the current logical HL^3 time value.

By associating a set of visibility attributes with each data item contained in a channel, it is possible to widen the visibility scope at later points in time. This can be used, for example, in formalisms where changes become immediately visible within the local context of the executing agent, but are published later to other agents (e.g. at the beginning of a new macro step).

Every `put()` operation on a channel leads to immediate distribution of the data within the whole cluster. This is performed by the ClusterCommunication service. If the visibility attributes refer to a future point in time, all cluster nodes will have a consistent view on this data, as long as the distribution is completed before the data becomes visible.

Abstract Machines Sequential components of a high-level formalism are mapped to Abstract Machines in HL^3. The task of each abstract machine is

– to indicate which transitions might be taken by the sequential component,
– to enable and disable flows according to the current state, and
– to indicate whether a flow phase may be started according to the abstract machines' local state.

While the concrete behavior of an abstract machine depends on the high-level formalism and the concrete high-level specification, the HL^3 framework defines a universal abstract interface for them: Operation `getTrans(id:amId)` returns the list of all transitions which might be performed in the current state. To this end, the abstract machine evaluates both the local static location data and the global state provided by channels. Since abstract machines represent sequential components, all high-level formalisms with a notion of nondeterminism require to select one out of several transitions which might be chosen in a specific state. Observe, however, that abstract machines do not perform this selection and never trigger associated actions. The former task is delegated to the Selector, the latter to the Scheduler.

After some enabled transition has been selected and its action performed, the associated abstract machine is notified by the scheduler (operation `notifyTrans()`) to trigger changes between locations within that abstract machine. The execution of actions associated with transitions will generally affect the global state encoded in channels. Therefore, the scheduler will call the `update()` operation of each abstract machine after transitions or flows have been performed. This operation initiates a new evaluation of all invariants and transition conditions applicable in the present state, possibly leading to a new set of enabled transitions within each abstract machine.

We expect that in every conceivable high-level formalism the execution of flows or transitions is mutually exclusive. Otherwise racing conditions might prevent the discrete change of observables due to simultaneous changes by flows. As a consequence, an abstract machine may indicate whether in the current state only transitions, only flows, or one of both may be performed. Depending on the local abstract machine state, the execution of flows may be disabled. This may happen in the case of high-level formalisms based on the maximal progress concept (sequential components with enabled transitions must fire) or allowing the definition of urgent transitions. Flows are also disabled if the abstract machine resides in a location whose invariant has just been violated, so that a transition becomes mandatory. The information whether flows may be performed is obtained via the `isFlowEnabled(id:amId)` operation.

A technical detail related to distributed execution of HL^3 components is indicated by the fact that the `getTrans(id:amId)` and `isFlowEnabled(id:amId)` operations have been declared on class level in Figure 6: While each abstract machine instance is created on a single cluster node only, where also their `init()`, `update()`, and `notifyTrans()` operations are scheduled, `getTrans()` and `isFlowEnabled()` may be called anywhere. This is motivated by the fact that the Selector should be able to call these operations from arbitrary nodes. The suggested implementation for abstract machines is to store the actual return values of these operations in associated channels, at the end of each `update()`

operation. Since channel data is consistently available on all nodes, `getTrans()` and `isFlowEnabled()` can return these values wherever they are called.

Selector The Selector is a centralized instance enforcing synchronization conditions for transition execution with respect to the high-level formalism. The abstract interface required by the HL^3 framework offers a `getSelection()` operation to the scheduler. It returns an indication whether a flow phase may be started and a (possibly empty) set of transition identifications with associated visibility sets. The transitions returned are the result of a selection procedure among all possible transitions offered by the abstract machines in their current state. In formalisms where transition sequences are supposed to be executed in zero time, the Selector keeps the same value for logical HL^3 time t_0 until all transitions within a zero-time step have been performed. Before the next flow phase, logical time is adapted to physical time. The actions associated with these transitions can be concurrently scheduled. Since actions derive pre-state from channels and change global state via channels as well, they may be triggered on arbitrary nodes and CPUs.

Note that even for the same high-level formalism it can be useful to apply different Selector instances: For application development, a selector will usually resolve nondeterministic transition selection – which may be allowed according to the high-level formalism – to deterministic execution sequences. In contrast to this, a simulation or testing system will require a selector which is capable of producing all transition schedules possible according to the high-level formalism.

Transitions Instances of Transition implement atomic state transformers, enabled by abstract machines. In order to support the partitioning of high-level model state into discrete *locations* and additional *variables*, transitions are equipped with a *trigger condition*, represented by Boolean function `condition()`. In contrast to high-level formalisms allowing condition expressions over global or local variables, HL^3 requires that condition functions retrieve state information from channels. The associated `get(id)` calls use the identification of the abstract machine owning the transition. Furthermore, each transition is associated with a (possibly empty) Action, implemented as a function reading the same channel data pre-state as the trigger condition, but also setting a post state via channels, to become visible at the point in time and for the indicated scope defined by the input parameter. Observe that HL^3 transitions are not equipped with any signal or event mechanism. We consider these as objects of higher-level formalisms, to be implemented in HL^3 by means of channels, the scheduler, and the selector component. This design decision could be revised as soon as the hardware platforms could provide semantically well-defined and fast signal mechanisms. However, our analysis of current PC or embedded controller hardware indicates that there is no such universally suitable mechanism for the embedded application domain. The concept of locations – if required by the high-level formalism – is encoded inside abstract machines. As a result of an action execution, abstract machines may be suspended, activated, or stopped, and the guards enabling/disabling flow execution may be set.

Flows Instances of Flow represent integration functions as a result of the discretization of time-continuous evolutions. The scheduler will activate all flows according to their specified frequency, provided that their `guard` attribute evaluates to *true*. The integration function `integrate(v:VisibilitySet)` is written in standard C/C++ syntax, but retrieves pre-state from channels instead of global or static local variables. Since flows are executed on behalf of their enabling/disabling abstract machines, they inherit the scope from the abstract machine. Based on current logical HL^3 time, the `integrate()`-operation reads the latest visible state and writes global data back to channels, to be published according to the visibility set v. Observe that HL^3 flows require integration functions which can be called with regular frequency. If the high-level formalism specifies flows by differential equations, these have to be solved using separate tools – such as Matlab – generating numerical libraries from given differential equations, for the purpose of discrete Δt integration.

Interface Modules A hardware abstraction layer conforming to the time-triggered system concept is provided by Interface Modules which are software components in one-to-one correspondence with hardware interfaces. Interface modules are scheduled with fixed frequency and perform an abstraction from raw data received on hardware interfaces to channel data and vice versa. When scheduled with the `poll(v:VisibilitySet)` operation, raw data is read from the interface and placed into the abstraction channel, using the visibility set passed by the scheduler with the call. If $t_0.t_1$ is the HL^3 time tick when the data has been received, the visibility set v ensures that the data will have been distributed to all cluster nodes before the earliest publishing time $t'_0.t'_1 > t_0.t_1$ defined by any visibility attribute contained in v. Conversely, each interface module retrieves the latest visible version of the associated channel data when scheduled and transmits this data item via driver and interface hardware. Observe that interface modules allow to use also interrupt-driven hardware devices in a time-triggered system: Interrupt handlers store the received data in intermediate buffers. Interface modules read the buffers when they are scheduled and publish the data via channels for the next periodic point in time, as required for the given interface in the physical constraints specification.

Scheduler Based on the synchronized cluster time introduced above, the Scheduler dispatches activities according to the following concept: Periodic scheduling of flows and interface polling is pre-planned at compile time, following the principles introduced for GIOTTO [HHK03]. For optimization purposes, activities a whose changes become effective at logical HL^3 time $(u_0.u_1)$ may be scheduled simultaneously with activities b to be published at $(v_0.v_1) \leq (u_0.u_1)$, if the pre-states for a are based on the visibility at an earlier time tick $(w_0.w_1) < (v_0.v_1)$. Since all activities as well as the cluster communication have bounded maximum length, each scheduler instance can determine when to start a new activity whose pre-state depends on the result of preceding ones, without the need to implement a commit protocol between cluster nodes.

5 HybridUML Semantics

The semantics of HybridUML specifications is defined by a transformation Φ_{HUML} to the Hybrid Low Level Language HL^3. As required by the HL^3 framework, a HybridUML specification is mapped to a selector and a collection of flows, transitions, and abstract machines (the aspects depending on architectural and physical constraints specifications are not considered here).

As described in Section 3, the main building block of a HybridUML specification is a set of basic agent instances. Their infrastructure for interaction is provided by connections between sets of variables or signals from different agent instances such that connected variables actually denote the same single shared variable or signal, respectively. For each connected set of variables or signals that cannot be extended we choose a unique Channel in terms of the low-level language HL^3 as representative of this set. This includes singleton sets, i.e. unconnected global variables and signals as well as local variables. As each variable and each signal is mapped to exactly one maximal set of connected variables or signals there is a function $chan : Var \cup Sig \rightarrow Chan_{hl3}$ that identifies the corresponding channel of a variable or signal. Signals are represented as channels carrying boolean data.

Transformation ϕ_{HUML} of Algebraic and Flow Constraints Algebraic and flow constraints of the HybridUML model are distributed over the modes of the basic agents. They define the set of HL^3-Flow instances by a mapping $flow_{hl3} : Flow \cup Alge \rightarrow Flow_{hl3}$: (1) The operation integrate() is provided by a mapping $proc : Exp \rightarrow op_{hl3}$ that defines an HL^3 operation of the form void op() for each HybridUML expression. The mapping of algebraic expressions is straightforward – variables v are mapped to local HL^3 variables that are read from or written to $chan(v)$, operators are mapped to corresponding HL^3 operators. Flow expressions are transformed by use of an appropriate (numerical) mathematical toolkit like Matlab. (2) For the boolean guard, a separate Channel is created. It is controlled by the abstract machine (described below) that corresponds to the flow constraint of the HybridUML model. (3) The frequency is obtained from the physical constraints specification. (4) A consecutive id is generated.

Transformation τ_{HUML} of Transitions Transitions in the HybridUML model connect the submodes of basic agents. The Transition instances t as well as the corresponding instances a of Action and c of Condition are given by
$trans_{hl3} : Trans \rightarrow \{(t, a, c) \in Trans_{hl3} \times Act_{hl3} \times Cond_{hl3} \mid t.a = a \wedge t.c = c\}$.

Transitions. A Transition instance is created that contains one condition and one action as described below. A consecutive transition id is generated.

Conditions. (1) The operation isTrueCondition() evaluates a boolean expression that is given by a mapping $bexp : Exp_{bool} \rightarrow op_{hl3}$. Similarly to $proc$, the mapping provides C expressions that directly implement the expression from the

HybridUML model; quantifiers on finite sets are realized by for-statements. Additionally, the (optional) signal is incorporated and treated as a conventional boolean variable. (2) The read set is created from the channels of the variables and the signal within the generated expression. (3) A consecutive condition id is generated.

Actions. (1) Similar to the integration operation of flows, action(...) is defined by *proc*; sending of signals is realized by sending true on the corresponding channel. The visibility parameter of action(...) is applied to the channels of the write set exactly as it is received (see below). (2) The writeSet consists of the channel identifiers that correspond to the written variables, i.e. the variables from action(...) that are on the left-hand side of assignments. (3) The variables from the right-hand sides define the channel identifiers in readSet. (4) A consecutive action id is generated.

Transformation α_{HUML} of Sequential Control Components Sequential control components are exactly the basic agent instances mentioned above. They are represented by instances of AbstractMachine: $am : Agent_{basic} \rightarrow AM_{hl3}$. Note that the arrangement of the basic agent instances to composite agent instances (through some levels of hierarchy) only provides the *distribution* and *renaming* of shared variables and signals, which is completely represented by the channel mapping *chan*.

An abstract machine defines the discrete behavior of a top-level mode, which in turn defines the discrete behavior of exactly one basic agent.

Data structure. The abstract machine defines a (recursive) data structure that maps to the hierarchical structure of the top-level mode, such that the Mode instances form a mode tree. Within the tree, the sequence of active submodes, beginning with the root mode, constitutes the mode configuration, i.e. the set of all currently active modes. Based on the mode tree, the data structure is defined in a straightforward way:
Mode A Mode consists of a set of control points of type ControlPoint, a set of submodes of type Mode, a set of transitions of type ModeTransition, a set of flow constraints of type FlowConstraint, and a set of invariant constraints of type InvariantConstraint. Additionally, a history variable points to the currently active submode. **InvariantConstraint** An InvariantConstraint contains a boolean function that evaluates according to a mode's invariant specification. **FlowConstraint** A FlowConstraint represents an algebraic or flow constraint from the HybridUML model. It references the low-level Flow according to $flow_{hl3}$. Particularly, the boolean guard is controlled in order to enable or disable the associated integration operation, depending on the current mode configuration. **ControlPoint** A ControlPoint contains outgoing transitions of type ModeTransition. Furthermore, a reference to its parent mode is included. The distinction between entry and exit control point is included as a flag. **ModeTransition** A ModeTransition connects a source and a target ControlPoint. It can fire, if

its Signal is present, and if its Guard evaluates to true. The firing of a transition (possibly) causes a discrete state change, therefore it is associated with the corresponding low-level transition of type Transition assigned by $trans_{hl3}$ that encapsulates this. Since a transition may affect the history of its containing mode, a reference to the Mode is also contained. **Signal** A Signal references a boolean flag that denotes if the corresponding signal is currently active. **Guard** A Guard contains a boolean function that implements the corresponding guard of a transition.

Additional entities exist in the data structure but are not described here. They are used for efficiency; for example, the set of currently enabled transitions is stored explicitly.

Data structure instantiation. A basic HybridUML agent defines an instance of the data structure: For each mode, an instance of Mode exists. Each Mode apart from the top-level mode is inserted into its parent's set of submodes. For its algebraic and flow conditions, FlowConstraint instances are created and linked to the appropriate Flow instances. Every invariant expression is represented by a boolean function `bool exp()` provided by *bexp*. A ControlPoint instance is created for every control point of the specification and linked with the corresponding Mode.

Each transition from the HybridUML model is represented by a ModeTransition instance which is linked to the ControlPoint instances that represent its source and target, respectively, as well as with its parent mode. The transition's parent mode is the mode for that it connects either two submodes or the mode itself with a submode. Every ModeTransition is equipped with a Guard, an optional Signal, and the Transition instance described earlier. The guard, the signal, and the action are taken from the HybridUML model in the same way as described for low-level Action and Condition instances, whereas the guard represents the Condition without signal. Figure 8 shows the data structure instance for the basic agent BrakePointController from Section 2.

(5) **readSetTransBrakingRequired** $\equiv$

$$
\begin{aligned}
&\{v \mid \exists i \in \{1..VTP_COUNT\} \bullet \\
&\quad v \equiv chan(vtpActive[i]) \vee v \equiv chan(brakePoint[i]) \vee \\
&\quad v \equiv chan(ra.vtp[i].v) \vee v \equiv chan(ra.vtp[i].x) \\
&\} \cup \{chan(x), chan(v)\}
\end{aligned}
$$

The HL^3 operations within guards and invariants (as well as actions and flows) operate on the HL^3 channels (representing the HybridUML variables and signals). Therefore, local HL^3 variables of appropriate types are used and distributed to the respective channels.

Execution semantics. The semantics of the abstract machine is given by the execution of the set of provided operations:
init() Executes the initialization step of the agent: The top-level mode's default entry point is entered. **notifyTrans(tr:transID)** Accepts the notification

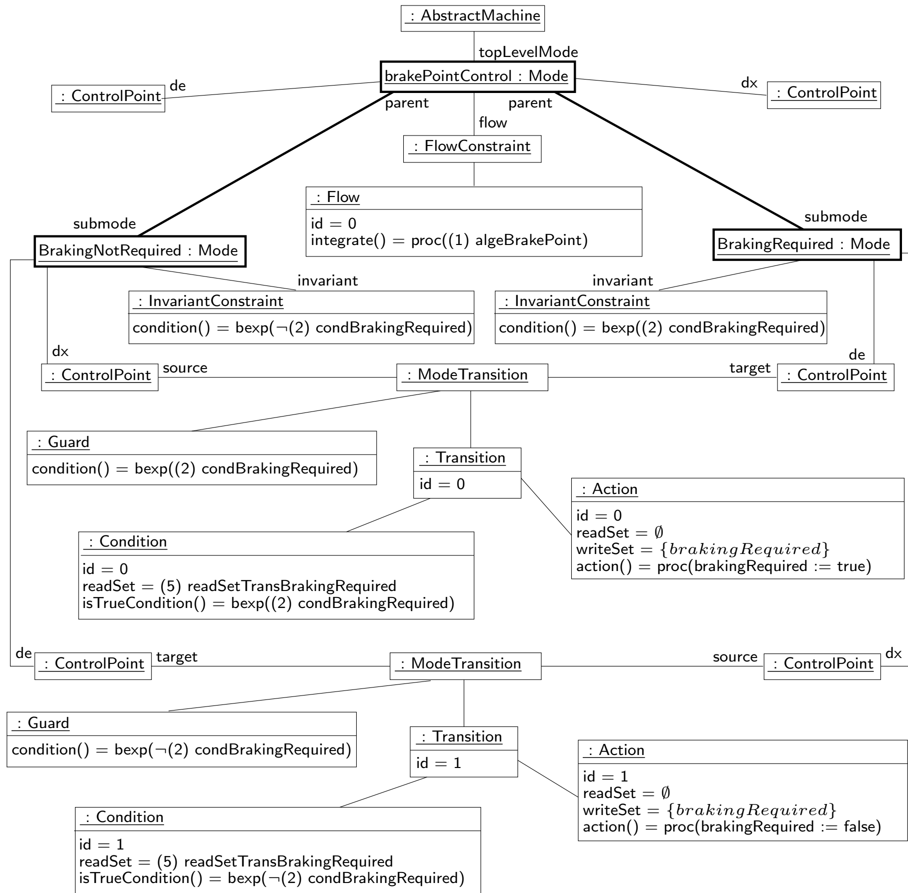

Fig. 8. Instantiated data structure of Agent BrakePointController. The mode tree is highlighted.

which transition was chosen and executed externally. Internally, the corresponding ModeTransition fires without executing its action, thus here it adjusts its internal state correspondingly by entering a new control point. **update()** Updates the internal data structure with respect to the current values of the corresponding channels: (1) A flag flowEnabled is set if this abstract machine is in a state that allows time to pass. This is given iff all invariants of the mode configuration are satisfied and there is no mandatory step. Therefore, the invariants of the mode configuration are recalculated. A mandatory step is a step that initially consumes a signal (i.e. that is triggered by a signal) or a step that does not start at a default exit point. (2) The set enabledTrans of enabled transitions is generated. (3) The set enabledTrans and the flag flowEnabled are written to respective

special channels, such that exactly these values are provided by subsequent calls of getTrans(amId) and isFlowEnabled(amId).

The precise behavioral description of the above operations is given by their implementation. For example, `notifyTrans(id:ID)` defines a discrete step:[1]

Abstract Machine The transition itself fires:

```
void AbstractMachine::notifyTrans (transID tr) { m_trans[tr].fire(); }
```

ModeTransition The transition fires by (1) leaving its source control point, (2) (optionally) resetting a consumed signal, (3) modifying its parent mode's history and (4) entering its target control point.

```
void ModeTransition::fire() {
  m_source.leave(); if (m_pSignal != 0) m_pSignal->setActive(false);
  writeHistory(); m_target.enter(); }
```

ControlPoint When a control point is left, its parent mode has no active control point anymore:

```
void ControlPoint::leave() {getParentMode().setCurrentControlPoint(0);}
```

When a control point is entered, it becomes the active control point of its parent mode:

```
void ControlPoint::enter() {getParentMode().setCurrentControlPoint(this);}
```

Mode The setting of a mode's current control point has several implications: (1) The mode's current control point is the new control point. (2) If the control point is the default entry, then the history is resumed recursively for the mode, if possible. (3) If for a leaf mode the control point is the default entry, then it is directly transferred to the default exit. (4) If the control point is the default exit, then all parent modes are also set to their default exits recursively. (5) If the control point is a non-default control point or a default entry that cannot resume a history, then the parent modes are recursively modified such that they have no current control point.

```
void Mode::setCurrentControlPoint(ControlPoint* pControlPoint) {
  m_pCurrentControlPoint = pControlPoint;
  if ((pControlPoint == m_pDe) && (m_pHistory != 0))
    m_pHistory->setCurrentControlPoint(m_pHistory->m_pDe);
  else if ((pControlPoint == m_pDe) && (m_leafMode))
    setCurrentControlPoint(m_pDx);
  else if ((pControlPoint == m_pDx) && (m_pParent != 0))
      m_pParent->setCurrentControlPoint(m_pParent->m_pDx);
  else if (m_pParent != 0) m_pParent->setCurrentControlPoint(0); }
```

In this way, we calculate the set of current control points which is required to determine the set of enabled transitions of the abstract machine. One of the following situations results: *Initialization* – the set consists of exactly one default entry of a non-leaf mode. There is no complete mode configuration, and only a transition that initializes the current mode can be enabled. *Stable* – the set consists of exactly the default exits of all modes of the current mode configuration from the root mode to the leaf mode. This is a (potentially) stable situation that may allow time to pass. Transitions of all hierarchy levels can be enabled. *Un-*

[1] The remaining definitions are omitted because of space restrictions.

stable – the set consists of exactly one non-default control point. Steps through non-default control points implicitely make up a compound step that must not be interrupted, therefore only continuation transitions may be enabled next.

HybridUML Simulation Semantics Definition Selector$_{HUML}$ For the coordination of flows, transitions, and abstract machines with respect to the *simulation* semantics of HybridUML, a customized Selector is provided. The selector defines an initialization operation init():bool:

The initial valuation of the channels has to be given by the environment, i.e. from outside the HL^3 specification. Initialization constraints result from the initStates of the agent instances within the HybridUML model. They control if there is an execution of the model for the given valuation. If all these constraints evaluate to true, initialization is completed successfully, otherwise unsuccessfully.

The operation getSelection(leftFlowPhase:bool): Selection is defined as follows:
(1) The logical HL^3 time $hl3time$ is adjusted. If the preceding scheduler phase was a flow phase, i.e. if $leftFlowPhase$, then setHL3Time(clustertime,0) of TimeService is activated and therefore the model time is updated to the cluster time. Since model time has increased, on every channel that represents a HybridUML signal, false is written, and thus signals are reset. Otherwise, after a transition phase ($\neg leftFlowPhase$), the t_1 component is incremented: setHL3Time(getHL3Time().t0,getHL3Time().t1+1);
(2) It is determined if a flow of time would be admissible for the complete model, which is denoted by the conjunction $flowEnabled = \bigwedge_{i=0}^{n} isFlowEnabled(am_i)$ for all abstract machines.
(3) The sets $tr_i = getTrans(am_i)$ of (identifiers of) enabled transitions are requested for all abstract machines.
(4) A set $tr = \{id \in \bigcup_{i=0}^{n} tr_i \mid \forall t_1 \in Transition, t_2 \in Transition \bullet ((t_1.id \in tr_k, t_2.id \in tr_l) \wedge (t_1 \neq t_2 \Rightarrow tr_k \neq tr_l) \wedge (t_1.a.readSet \cup t_1.c.readSet) \cap t_2.a.writeSet = \emptyset)\}$ is chosen non-deterministically. Thus, a set of transition identifiers is chosen that (1) contains up to one transition identifier per abstract machine and that (2) contains only identifiers of transitions that are independent of each other, i.e. every possible execution sequence of these transitions is allowed. Note that $n = |tr| > 1$ is just an optimization of n successive transition phases selecting one transition each. If $\neg flowEnabled$, $tr \neq \emptyset$ is preferred, otherwise the HybridUML model is deadlocked.
(5) If $tr \neq \emptyset \wedge flowEnabled$, then an "almost" non-deterministic choice is made between transitions and flows: Since the scheduler always executes possible transitions, as long as there is enough time left before the next flow phase, $tr = \emptyset$ may be enforced.
(6) A visibility attribute att_v is created that satisfies $att_v.t.t0 = hl3time.t0 \wedge att_v.t.t1 = hl3time.t1 + 1$, and which has an unrestricted scope $att_v.scope$, i.e. every abstract machine or interface module that reads the channels of the written variables is included. The visibility set $\{att_v\}$ is attached to every transition: $trv = \{(t, v) \in Transition \times VisibilitySet \mid t.id \in tr \wedge v = \{att_v\}\}$.

(7) Finally, a Selection s with $s.isFlowEnabled = flowEnabled$ and $s.trans = trv$ is returned by update.

6 Conclusions

We have introduced HybridUML, a novel specification formalism for the description of hybrid systems. HybridUML was defined as a profile extending UML 2.0. The main intention of this approach is to facilitate the understanding of the formalism for users already familiar with the UML and to utilize existing UML tools for the development of hybrid specifications. The "look-and-feel" of HybridUML was illustrated by means of a case study describing a distributed radio-based train control system. The semantics of HybridUML has been obtained via transformation into the Hybrid Low-Level Language framework HL^3, thereby obtaining semantically well-defined programs which can be executed in hard real-time.

When compared to GIOTTO [HHK03], our HL^3 framework differs in the following aspects: (1) The HL^3 channel concept – corresponding to GIOTTO ports – explicitly supports visibility time stamps and scope. We regard these mechanisms as very helpful for ensuring consistent data views on different cluster nodes in time-triggered systems. (2) The HL^3 framework has been explicitly designed to facilitate the automated generation of compilation targets from higher-level formalisms. This is reflected by the pre-defined roles and interfaces of abstract machines and selector. (3) GIOTTO tasks have a low granularity, corresponding to single flows, transitions or transition collections emanating from the same location. Transitions between locations as, for example, between hierarchic states of a statechart, have to be modeled by GIOTTO mode switches disabling/activating "task vectors". In contrast to this, the HL^3 abstract machines encode behavioral models of complete sequential agents; only the flows and actions are separated from the abstract machines, in order to optimize scheduling.

Our current investigations related to semantic issues focus on the respective advantages and tradeoffs presented by interleaving semantics versus "true parallelism" interpretations for transitions executed in the same "zero-time phase". The interleaving semantics as defined in this paper and suggested in [AGLS01] is compatible with the rely-guarantee verification method [dR$^+$01, pp. 447] developed for shared-variable concurrency. Therefore its utilization gives us the advantage of well-understood formal concepts and proof theories. A major drawback is the fact that interleaved transition execution – while being perfectly well-suited on single-processor platforms – cannot easily be distributed for parallel execution on several processors: The action of one transition may invalidate the firing condition for another transition, so in the worst case – if the write sets of the actions for all available transitions overlap with the read sets of all their conditions or actions – it is mandatory to execute one transition at a time. In contrast to this, Statecharts semantics [DJHP98] defines truly parallel execution rules for transitions simultaneously enabled in parallel components: All transitions available at the beginning of a macro step have the same view on the state

components within their scope and may fire simultaneously. Their state changes become visible in subsequent micro or macro steps. Obviously, these execution rules are well-suited for multi-processor scheduling environments. However, this advantage has to be paid by increased verification complexity, as, for example, reflected by the problem of racing conditions occurring due to simultaneous changes to the same variables in the same step.

Apart from facilitating the development of hard real-time target systems, our transformation strategy supports automated testing of hybrid systems. Here, HybridUML agents are used both for the specification of the system under test and for the description of environment behavior to be simulated in specific test executions. While the transformation from HybridUML into the abstract machines of HL^3 is the same for target system development and testing, different selector instances are used in these two situations: For developing target systems, the semantic freedom which could be exploited by the selector in the non-deterministic choice among possible transition interleavings, as well as the decision when to trigger enabled, but non-urgent transitions, should be resolved in a deterministic way which can be relied on to meet all periodic schedules. In contrast to this, a selector for testing hybrid systems may apply strategies to simulate the greatest possible variation of environment behavior, in order to increase the structural coverage of the system under test and to check its robustness. A more detailed description of testing aspects is currently under preparation [BBPT04].

The authors would like to emphasize that HL^3 is not just an experimental runtime environment for research purposes. Its current version is used for embedded systems testing of controllers for the Airbus A380 aircraft family [VS04]. Test engines operate with cluster configurations consisting of 3, 5, or more multi-CPU PC nodes.

Acknowledgements. The authors are indebted to Aliki Tsiolakis for her stimulating comments and suggestions on HL^3, its semantics, and implementation. Christof Efkemann and Kai Thomsen did a formidable job in their development of a Linux kernel patch for CPU reservation and interrupt relaying.

References

[AGLS01] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, *LNCS* vol. 2034, pp. 33–48, 2001.

[BBB+99] T. Bienmüller, J. Bohn, H. Brinkmann, U. Brockmeyer, W. Damm, H. Hungar, and P. Jansen. Verification of automotive control units. In *Correct System Design,LNCS* vol. 1710, pp. 319–341, 1999.

[BBH+04] K. Berkenkötter, S. Bisanz, U. Hannemann, J. Peleska, and A. Tsiolakis. The Hybrid Low Level Language HL^3. Technical Report 34, Technologie Zentrum Informatik TZI, Universität Bremen, to appear July 2004.

[BBHP04] K. Berkenkötter, S. Bisanz, U. Hannemann, and J. Peleska. The HybridUML Profile for UML 2.0. Technical Report 32, Technologie Zentrum Informatik TZI, Universität Bremen, June 2004.

[BBPT04] K. Berkenkötter, S. Bisanz, U. Hannemann, J. Peleska, and A. Tsiolakis. Automated Test Data Generation for Hybrid Systems.

[DFG] Priority Programme Software Specification – Integration of Software Specification Techniques for Applications in Engineering. http://tfs.cs.tu-berlin.de/projekte/indspec/SPP.

[RTAI03] Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. RTAI homepage. http://www.aero.polimi.it/ rtai/about/index.html, 2003.

[DJHP98] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. *LNCS* vol. 1536, pp. 186–238, 1998.

[dR+01] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhneche, M. Poel, and J. Zwiers. *Concurrency Verification*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, April 2001.

[Hen96] T. A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pp. 278–292. IEEE Computer Society Press, 1996.

[HHK03] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91, pp. 84–99, 2003.

[HL94] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *IEEE Real-Time Systems Symposium*, pp. 120–131. IEEE Computer Society, 1994.

[KMP00] Y. Kesten, Z. Manna, and A. Pnueli. Verification of clocked and hybrid systems. *Acta Informatica*, 36(11):836–912, 2000.

[Kop97] H. Kopetz. *Real-Time Systems – Design Principles for Distributed Embedded Applications*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1997.

[Lab04] FSM Labs. RT-Linux homepage. http://www.rtlinux.org, 2004.

[OMG03a] OMG. UML 2.0 Infrastructure Specification, OMG Adopted Specification. http://www.omg.org/cgi-bin/apps/doc?ptc/03-09-15.pdf, September 2003.

[OMG03b] OMG. UML 2.0 Superstructure Specification, OMG Adopted Specification. http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.pdf, August 2003.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language – Reference Manual*. Addison-Wesley, 1999.

[RRS03] M. Rönnkö, A. P. Ravn, and K. Sere. Hybrid action systems. *Theoretical Computer Science*, 290:937–973, January 2003.

[VS04] Verified Systems. RT-Tester 6.x – User Manual. Technical Report Verified-INT-014-2003, Verified Systems International GmbH, Bremen, 2004.

[ZRH93] C. Zhou, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid real-time systems. In *Hybrid Systems*, pp. 36–59. The Computer Society of the IEEE, 1993. Extended abstract.