HybridUML Profile for UML 2.0

Kirsten Berkenkötter Stefan Bisanz Ulrich Hannemann Jan Peleska

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

Abstract

In this article, a new UML extension for the specification of hybrid systems, where observables may consist of both discrete and time-continuous parameters, is presented. The construction uses the profile mechanism of UML 2.0 which is the standard procedure for extending the Unified Modeling Language. Whereas hybrid modeling constructs are not available in standard UML, several specification formalisms for this type of system have been elaborated and discussed in the formal methods communities. As a basis for the new UML profile, the authors adopt one of these formalisms – the CHARON language of Alur *et. al.* – already possessing several attractive features for modeling embedded real-time systems with hybrid characteristics. As a result, the profile inherits the formal CHARON semantics, so it offers the possibility for formal reasoning about hybrid UML specifications. Conversely, CHARON is associated with a new syntactic representation within the UML 2.0 world, allowing to develop hybrid specifications with arbitrary CASE tools supporting UML 2.0 and its profiling mechanism. The "look-and-feel" of the profile is illustrated by means of a case study of an embedded system controlling the cabin illumination in an aircraft. The benefits and weaknesses of the constructed hybrid UML profile are discussed, resulting in feed-back for the improvement of both UML 2.0 and the CHARON formalism.

1 Introduction

In this paper, we suggest an extension of UML 2.0 for hybrid systems modeling. A real-time system is called *hybrid* if it processes *time-continuous* variables in addition to discrete-range parameters. The specification of (piecewise) continuous evolutions for variable values over dense time 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.

While UML 2.0 [OMG03a, OMG03b] provides language constructs for a large variety of systems and behaviors, elements for hybrid systems modeling have not been considered as essential for the Unified Modeling Language: Real or rational numbers are not established as primitive types [OMG03a, 6.1], so the constraints governing the time-continuous change of values obviously cannot be described, either.

The need to extend UML has already been anticipated in "classical" UML [JRB99, p. 104, p. 531], in particular, for domain-specific adaptions. New features may be added by creation of *profiles*, i.e., addition of new packages to the reference meta model. As the main tool for extension construction, *stereotypes* are used. Following the revised profile construction concept of UML 2.0 [OMG03a, 7.1], we generate a hybrid systems extension for UML 2.0.

In "classical" UML, 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 [JRB99, pp. 105] this approach is motivated by the fact that the semantic interpretation of specification constructs depends on the specific project context. Precise semantics is mandatory as soon as

- unambiguous meaning (e.g. in order to avoid legal problems),
- formal reasoning about the specification (model checking, elaboration of proofs),
- simulation of the specification

is required. For associating precise semantics, basically two approaches are available: The bottom-up approach lifts the meaning of UML expressions (actions, constraints etc.) from the underlying programming language and

– if this is still not sufficiently precise – from the underlying machine code and hardware model. This approach is suitable for the development of simulations, but usually infeasible for the development of proof systems for mathematical reasoning. In contrast to this, the top-down approach starts with a specification language already associated with a formal semantics, and the meaning of these language constructs is "lifted" to the associated constructs in UML syntax along with the profile construction process. In the present paper, we follow the top-down approach, starting with a formal language explicitly developed for hybrid systems modeling.

Instead of creating a new hybrid systems modeling formalism along with the UML profile, our objective is to show how an *existing* formal specification language may be imported into the UML by means of profile construction. As a result, the original formalism is associated with a new syntax conforming to the UML paradigm and usable within UML tools supporting the profile mechanism. This may be considered as an advantage, if UML syntax has already been established within development and verification teams, and if existing CASE tools supporting UML should be re-used for hybrid systems development.

In the formal methods community, hybrid systems have been thoroughly investigated during the last decade. From the large variety of competing formalisms we mention the Duration Calculus [ZRH93] and Hybrid Automata [Hen96]. While the former introduces new notation for logical formulae, the latter extends state machine notations for the specification of time-continuous behavioral aspects.

When selecting an appropriate candidate, it is promising to choose a language where a number of syntactic features and their intuitive meaning already correspond quite closely to existing UML concepts. In this respect, Hybrid Automata are more suitable than the Duration Calculus, since they are based on state machine descriptions comparable to the Statecharts variant of UML. However, according to the authors' understanding, Hybrid Automata have been designed as a scientific research language, in order to investigate and demonstrate the possibilities and limitations of model checking strategies for dense-time/continuous-value systems. For practical utilization in sufficiently large "real-world" developments, the flat structure of Hybrid Automata will often lead to specifications of unmanageable size and complexity. This problem has motivated the development of the CHARON formalism [ADE+03] where specifications are constructed as networks of communicating agents, internally modeled by hybrid automata with a hierarchical structure. CHARON only admits communication via shared variables; however, this communication paradigm is adequate for "small" embedded systems with simple underlying operating systems and tight timing restrictions. Therefore we adopt CHARON as the starting point for our profile construction.

In Section 2, we give a brief overview on CHARON and its basic mechanisms for modeling hybrid systems. Section 3 introduces the profile concept of UML 2.0 in general, before we describe the new hybrid systems profile in Section 4. Due to the usual space limitations, the full version of the profile could only be included as appendix. Section 5 presents a case study from the field of embedded avionics systems: We give a simplified hybrid specification of continuous illumination intensity control, as it is used today in modern aircrafts. Section 6 concludes with a discussion of the new profile and resulting suggestions for both UML and CHARON.

2 CHARON at a Glance

As embedded systems which interact with the physical environment are increasingly used in safety-critical application areas, e.g., for automotive control, avionics, telematics, chemical process control systems, it becomes more and more important to guarantee their safe functioning and improve the entire development process with the focus on this issue. A rigorous formal approach is crucial here as such systems are notoriously complex.

Combining discrete state machines with continuous behavior, hybrid systems [ACH+95, Hen96] have been successfully used to model a large number of applications in areas such as real-time software, embedded systems, and others. Basically, it is a state-based formalism augmented by real-valued variables which may continuously evolve over time. The discrete behavior is given as a labeled transition system, typically in guarded-command notation, allowing shared-variable communication and synchronization over transition labels. The continuous behavior is typically specified per control-state by differential (in-)equations.

Among the languages developed for the modeling of hybrid systems, CHARON [AGLS01, ADE+01, ADE+03] provides formal semantics which are necessary for reasoning about the model. The language allows specification of architectural as well as behavioral hierarchy and supports exception handling through so-called group transitions.

In CHARON, the building block for describing the system architecture is an *agent* that communicates with its environment via shared variables. The language supports the operations of *composition* of agents to model concurrency, *hiding* of variables to restrict sharing of information, and *instantiation* of agents to support reuse [ADE+01]. Formally, an agent consists of a set of variables V, partitioned into local and global variables, a set of initial states, and a set of *modes*, where a mode is basically a hierarchical state machine which describes the flow of control inside an agent. A mode consists of a set of submodes, a set of variables, a set of *entry control* *points*, a set of *exit control points*, a set of transitions, and a set of constraints. Transitions of a mode always start either at an entry point of that mode or at an exit point of a submode, and they lead to an exit point of that mode or to an entry point of a submode, thus there are no inter-level transitions. Constraints are given as differential (in-)equations, as algebraic constraints, and as invariants, they are evaluated hierarchically at the various levels to simplify the description of the system.

Reflecting the double-layered modeling, the definition of the semantics of an agent is based on the definition of the semantics of modes. As typical for hybrid systems, there are basically two ways of acting: either some transition is taken, where variables are changed instantaneously, or some time passes, where the system resides in the same control location (as modeled by a mode) and the continuous variables change over time according to their specified constraints.

The concept of clearly specified entry and exit points allows for a clean interface definition of a mode, a requirement for further refinement in the development process. Control points also facilitate a simple concept to model preemption, leading out of a mode with inner hierarchical structure as *group transitions*. While it would be possible to define semantics for this model based on a transformation to non-hierarchical hybrid automata, one of the main purposes of CHARON is its application for tool-based simulation. As such a transformation would change the representation drastically, an alternative semantics is defined by means of a closure operation which adds a number of transitions to a given mode which reflect implicit activities of the system. This operation includes the introduction of a *history* variable that is local to each mode, which records the currently active submode. This is required to resume a computation at some location that was left via a group transition. As a conceptual decision of CHARON, an interrupted mode will always resume his activity in the location that was active when control was transferred.

When a mode is executing a continuous step, i.e., when time passes, one has to keep in mind that the hierarchical state machine as a whole is acting. For time-delay steps, 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. The relation $\mathcal{R}^C$ defining the change of values over time for a mode is thus defined in terms of the constraints of that mode and the relations associated with its (active) submodes.

For each pair of control points c_1 and c_2 of a mode, a relation $\mathcal{R}^D_{c_1,c_2}$ is defined describing the path from c_1 to c_2 by the state changes induced along this path (no time passes taking a discrete transition). For a leaf mode this is identical to the transitions allowed in that mode, for a mode that contains submodes, such a path is composed out of transition steps of the mode itself and of steps which are taken within a submode from an entry point of that submode to an exit point of it. The relation $\mathcal{R}^D_{e,x}$ describes the state changes induced along a path from an entry point e of the mode to an exit point x and is called a *macro-step*, as this may include transitions of the mode itself as well as steps of its submodes.

From the viewpoint of a mode within the hierarchical system, there are three possibilities to change the state of its variables, i.e., to take a step within the system:

- a discrete step, taken by the mode itself, is some macro-step out of $\mathcal{R}^D_{e,x}$, leading from an entry point e of that mode to an exit point x.
- a continuous step, where time passes according to $\mathcal{R}^C$ while control remains in the mode.
- an environment step, representing activity of the *same* agent in a different location. This leads from an exit point to an entry point and may change all but the private variables.

An execution of a mode is described by a sequence of pairs (c, σ) with c a control point and σ a valuation, where transitions between subsequent pairs are labeled to indicate what kind of step caused the change between them.

A *primitive* agent has a single top-level mode, *composite* agents are constructed by parallel composition of other agents and have many top-level modes. The semantics of a primitive agent are derived directly from the semantics of the top-level mode which has exactly three control points: some *init* entry point, a default entry point *de* and a default exit point *dx*, which precises the possible executions of the mode to consist of an initial discrete macro-step from $\mathcal{R}^D_{init,dx}$, and next a continuous step. The environment step that follows now is an identity mapping, as there is no hierarchical environment to be considered, moving from *dx* to *de*. (The top-level mode is always active.) Then we have a discrete macro-step from $\mathcal{R}^D_{de,dx}$, before taking the next continuous step.

These semantics correspond already with common definitions for hybrid automata models [ACH+95], when seen as a closed system that does not interact with its environment, as one can ignore the environment steps in this case, resulting in an alternating sequence of discrete and continuous transitions for a top-level mode. But as agents in CHARON are supposed to interact with each other, the above explanatory view is extended to fit *open* systems. An interleaving model is employed to capture the effect of state changes due to discrete changes by transitions of a possibly unknown environment, as agents communicate via shared variables. Therefore another type of environment steps is introduced on this level, now reflecting discrete state changes caused by the environment of the agent. For time passing steps it is obvious that all components in a concurrent hybrid system have to participate in such a step. The control location information of a mode remains local information for the enclosing agent, and an execution of a primitive agent is defined by a sequence of valuations, obtained by a projection from the execution of the top-level mode to a specified set of variables.

For a composite agent, executions of the constituent agents are merged as follows: first, all (top-level) modes go through their initialization step, no continuous step is possible until all modes have completed this. Then we have either a time delay step, with all modes taking part in, a discrete step of one of the modes, or a discrete environment step, where global variables may change while local variables may not. As composition in parallel is allowed only on the level of agents, these semantics are suited also for compositional refinement proofs, i.e., where a refinement (defined by trace refinement) of one component can be proven to preserve the specification of the system [AGLS01].

As CHARON models have a precise semantics, they can be subjected to a variety of analysis techniques, e.g., simulation, reachability analysis, and predicate abstraction [ADE$^+$03]. The toolkit provided for CHARON contains a graphical user interface, a type-checker, a simulation tool and a plotter [CHA].

3 UML 2.0 Profiles

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.

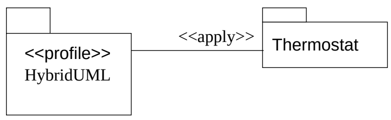

Figure 1: Profile Application by Package

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 (see Fig. 1). The profile itself is a package and therefore depicted like this with the keyword ≪*profile*≫ above its name.

As described before, stereotypes extend elements of the metamodel in use, i.e. they extend a class of the metamodel. Information can be added but not taken away as the model has to be valid without the profile. Generalization of stereotypes is allowed. In the profile, the keyword ≪*stereotype*≫ marks the extended element while its name is given below (see Fig. 3). In the model that applies the profile, the name of the stereotype in guillemets is used as a keyword (see Fig. 4).

An extension is always binary, i.e. a stereotype is dependent on exactly one element of the underlying metamodel. It is depicted by an arrow with filled arrowhead. The extension can be marked as {*required*}, i.e. the stereotype is always created if an instance of the extended class is created. In other words, the extension is mandatory in this case.

4 HybridUML

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. This fact has not changed with UML 2.0 [Ber03]. Therefore, another solution must be found for using UML in the real-time domain.

HybridUML is a UML 2.0 profile that is based on CHARON. It takes a subset of UML, modifies it according to CHARON and gives it precise semantics. Therefore the most important constraint on applying the HybridUML profile is using only the model elements specified in it. Other elements are not covered in the profile including sequence diagrams, activity diagrams, and so on.

In the appendix, we develop the HybridUML profile by listing the involved UML constructs each with a full systematic description of their use, attributes, associations, constraints, semantics, and notation. For brevity of the present paper we restrict ourselves to an overview on the relevant elements.

4.1 PrimitiveType

CHARON uses typed datatypes. In contrast to Integer, String, and Boolean, real numbers are not covered in UML. We extend PrimitiveType for getting a Real datatype. We also need analog real numbers, i.e. real numbers whose value is varying with passing time. The way the value is changing is given in an expression of the type RTExpression that is described in the next section. AnalogReal variables and RTExpressions are attached to one another in Modes. AnalogReal is a generalization of Real. AnalogReal numbers are used in statecharts for describing flow conditions and invariants.

4.2 Expression

For describing the evaluation of AnalogReal variables, different expressions are needed, i.e. differential expressions and algebraic expressions. Invariant expressions are needed for defining state invariants. RTExpression is an instance of Expression (see Fig. 2). It defines mathematical and logical terms that may be dependent on time. RTExpression is an abstract metaclass that cannot be instantiated.

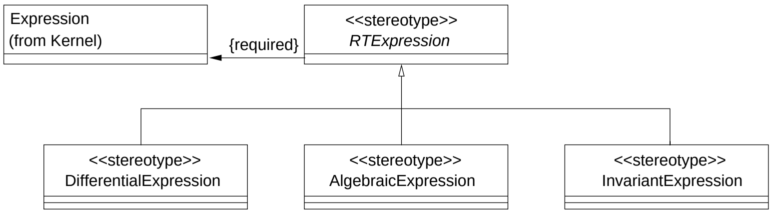

Figure 2: Stereotypes for RTExpressions

The real-time expression is given in the attribute *symbol* 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, which are

- AlgebraicExpression: as a generalization of RTExpression, it describes algebraic terms independent on time.
- DifferentialExpression: as a generalization of RTExpression, it describes differential terms dependent on time.
- InvariantExpression: as a generalization of RTExpression, it describes logical terms used for modeling invariants a variable must fulfill.

4.3 Constraint

For describing the flow of analog variables in CHARON, constraints are used. These are given in Modes that use the corresponding variable.

RTConstraints are UML constraints which are restricted in order to describe an RTExpression.Therefore, DifferentialExpressions, AlgebraicExpressions, and InvariantExpressions can be used. RTConstraint is an instance of Constraint. It holds an RTExpression. DifferentialExpressions can be attached to AnalogReal variables, AlgebraicExpressions to Real and Integer 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 (see Fig. 7).

4.4 Time

For modeling time, we need clocks. This is done by using a variable of type AnalogReal that uses a DifferentialEquation for modeling the flow of time. Therefore we inherit from AnalogReal to get a clock. We do not use the UML 2.0 time model as it has no formal semantics and it is not powerful enough for our purposes. A Clock is modeled by an AnalogReal variable. 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.

4.5 Structure

4.5.1 Global Variables

In CHARON, Agents talk to each other exclusively over global variables. They are visualized in the same way as UML 2.0 Ports, which are linked by connectors. A VariablePort is an instance of Port. It has only required and provided VariableInterfaces. VariablePorts are always service ports and behavior ports, i.e. they are interaction points of Agents and connected to a state machine called Mode. Each VariablePort owns exactly one VariableInterface.

VariableInterface is an instance of Interface. A VariableInterface is associated to one global variable that must be of type PrimitiveType. VariableInterfaces own exactly one property, its name is given as the name of the VariableInterface, respectively as the name of the VariablePort that owns the interface. Consequently, the properties of connected interfaces must have the same value. Required VariableInterfaces correspond to read access and provided VariableInterfaces correspond to write access with respect to CHARON. Write access means read/write access.

VariableConnectors are instances of Connector. They link VariablePorts whose required or provided interfaces must be instances of the same VariableInterface, i.e. they mirror the same global variable. VariableConnectorEnds are instances of ConnectorEnd. They are the ends of VariableConnectors. They are always attached to VariablePorts.

Notation

VariablePorts are depicted like ports, i.e. a rectangle on the boundary of the owning classifier. Instead of visualizing the attached interfaces in lollipop-notation, a required interface is a white filled rectangle and a provided interface is a black filled rectangle (see Fig. 5). As every port is connected to a Mode, the state symbol that indicates behavior ports will be omitted. In class diagrams, only the variable owned by the VariableInterface of the port will be shown (see Fig. 4). VariableInterfaces are only visualized in combination with ports. VariableConnectors are solid lines between ports, while VariableConnectorEnds do not have a notation.

4.5.2 Agent

Agents are the main element for modeling structure with CHARON. They consist of a set of variables, a set of Modes, i.e. statecharts, and a set of initial states. Agents can be composed of other Agents, respectively Agent instances.

In HybridUML, Agents are stereotypes of classes. They consist of VariablePorts, 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).

Description

An Agent is an instance of Class (see Fig. 3) that can own an internal structure (see Fig. 5). The internal structure consists of Agent instances. Agents communicate by VariablePorts and VariableConnectors.

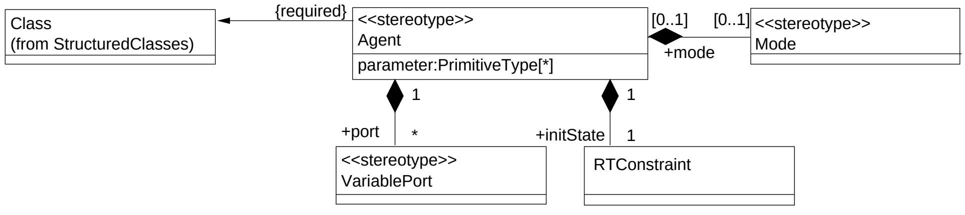

Figure 3: Stereotype for Agent

Semantics

We apply the semantics of the modeling language CHARON [AGLS01, ADE$^+$01] to the Agents in our HybridUML profile. See Section A.5.1 in the appendix for a detailed explanation. We distinguish *primitive* Agents which are not nested and own a single top-level Mode, and *composite* Agents which are composed from sub-agents 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 TM define the behavior of the system. The semantics of an Agent are defined by the (trace) behavior of its top-level Modes, constructed from the respective relations R^C on the continuous behavior and R^D for the discrete transitions of a Mode (details on the semantics of Modes are presented in Section A.6.1 of the appendix). 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 updates to the variables of the Agent. Continuous steps are performed by all Modes, while discrete steps are performed by one Mode at a time. Before each discrete step, there is an environment step which leaves all the local variables of all top-level Modes intact. These are needed to model open systems appropriately.

Operations on Agents in CHARON are the composition of two Agents $A_1 \| A_2$, application of a hiding operator, and renaming of the variables of an AgentInstance. These operations can be reflected already in the representation of an Agent's internal structure in a composite strucure diagram.

Notation

Agents are depicted like UML classes with internal structure. In a class diagram, the internal structure is visualized as aggregated classes (see Fig. 4). The parameter list of each Agent is given behind its name in parentheses in the first compartment of the class symbol.

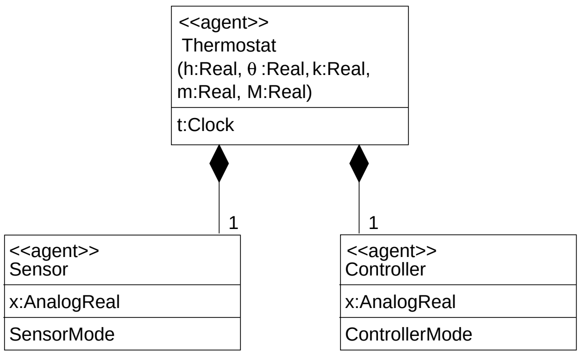

Figure 4: Notation for Agent in Class Diagram

VariablePorts and their included VariableInterfaces and variables are given as attributes in the second compartment of the class symbol. In the class diagram, only the name and type of the included variable is given.

Optionally, in the third compartment of the class symbol 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. The name of the Agent is given in the upper left corner with the keywort *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 (see Fig. 5).

Agent instances are visualized as objects in composite structure diagrams (see Fig. 5). 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, 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 (see Fig. 7).

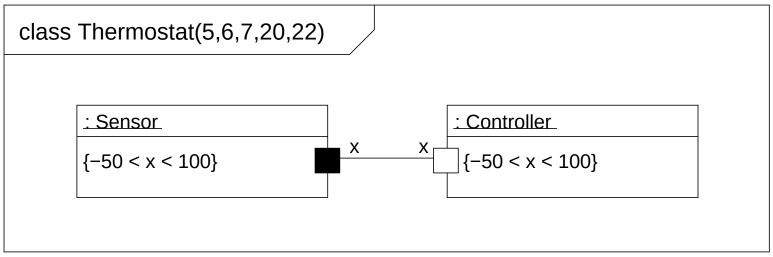

Figure 5: Notation for Agent in Composite Structure Diagram

4.6 Behavior

4.6.1 Modes

In CHARON, Modes are used for describing the behavior of an Agent. These are hybrid state machines. In our profile, Mode is an instance of StateMachine describing an Agent's behavior. 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. Every Mode has a default entry point de and a default exit point dx.

A Mode that is not contained by any other Mode is called top-level Mode. Each top-level Mode has a single non-default entry point called *init* point, and no non-default exit point. A Mode that is contained by another Mode is called Submode. A Mode without Submodes is called leaf Mode.

Top-level Modes are connected to an Agent. They use the global variables defined in this Agent. Analog variables are updated according to constraints while the state machine is in a Mode. Discrete variables are only updated when a transition is taken. Modes can have parameters for better scalability. Preemption is modeled by using the default exit point dx as source of a group transition.

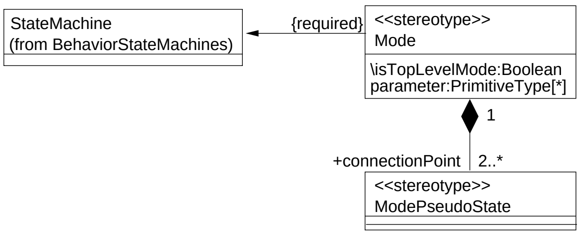

Figure 6: Stereotype for Mode

As Modes are hierarchical hybrid automata, the semantics as given in [AGLS01, ADE+01] consists of discrete and continuous steps taken by a mode, where additional environment steps are observable for submodes that may be inactive. A Mode can engage in discrete or continuous behavior, by either taking some transition and changing variables accordingly to its action, or by residing in a mode and letting time pass, updating the analog variables according to their flow constraints. See Section A.6.1 in the appendix for a detailed explanation and the derivation of a trace semantics which provide the base for defining the behavior of the enclosing agent.

Notation

Modes are visualized the same way as UML 2.0 StateMachines (see Fig. 7). Parameters are given behind the name of the Mode in parentheses. The invariant is marked *inv*, the flow conditions with *flow*. As both are constraints, they are given in brackets.

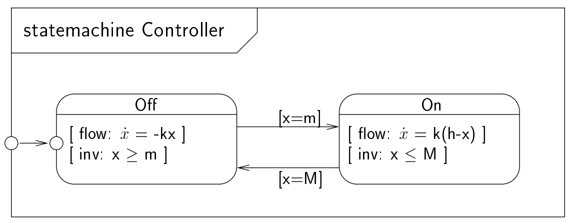

Figure 7: Notation for Mode

4.6.2 Other Elements

ModePseudostate is an instance of Pseudostate. ModePseudostates are used as entry and exit points for Modes. Further, they connect multiple transitions into more complex ones. A ModePseudostate is either entryPoint, exitPoint, defaultEntry, defaultExit, or junction according to ModePseudostateKind.

entryPoints are depicted as small circles on the border of a Mode (see Fig. 7) with an optional name attached to it while exitPoints are depicted in the same way but as small solid black-filled circles. defaultEntry points are not depicted explicitly as every Mode has exactly one. Transitions to the defaultEntry point end at the boundary of the Mode. The same holds for defaultExit points. They are not depicted explicitly and transitions starting there start at the boundary of the Mode. junction points are depicted as small black-filled circles inside Regions.

ModeRegion is an instance of Region. It contains ModeStates and ModeTransitions. Each Mode consists of one region. Orthogonal regions are not allowed, and hence, there is no notation for region. They are just the space inside a Mode.

ModeState is an instance of State. Each Mode consists of ModeStates. ModeStates are always Submodes, their semantics are given already in Mode, and they are visualized in the same way as UML 2.0 States (see Fig. 7).

ModeTransition is an instance of Transition. It connects a target and a source ModePseudoState. ModeTransitions are taken due to guard constraints. A ModeTransition may have an associated effect that updates some variables.

A transition that originates at a default exit point of a Mode is called group transition. Group transitions interrupt the execution of that Mode. After that, the Mode must be entered again through the default entry point to resume the execution of the Mode. ModeTransition is depicted by an arrow with open arrowhead (see Fig. 7). The guard constraint is given in brackets. The updateAction is separated from the guard by a slash. The default transition from de to dx of each Mode is not visualized.

In UML 2.0, Transitions can only have Activities as effect. We therefore need an Activity that updates variables as this is the only thing we do when performing transitions. ModeTransitionActivity is an instance of Activity which updates variable values of Agents.

5 A Case Study: Aircraft Cabin Illumination

In this section, a simplified real world application is modeled using the HybridUML Profile – the cabin illumination function that controls the illumination in the cabin of modern aircrafts. In [HP95] a similar system is formally specified using the specification language Z. There, illumination intensity essentially changes discretely, whereas the variant used in this article is designed for more modern equipment and involves illumination scenarios that continuously adapt the illumination intensity according to (linear) functions. The case study in this section focusses on this particular aspect of the illumination function evolving from a CHARON model[1] for simulation purposes.

The cabin is divided into a small number of cabin zones that contain several seat rows each. Furthermore, the cabin contains of seat columns that subsume the seats that are located at the windows, at the aisles, and in the center, respectively. The cross product of the zones and columns defines a set of locations that are controlled individually by the illumination function. Output is the respective illumination intensity.

There are different user-controlled inputs to the illumination function that trigger intensity changes. The inputs bright and off cause discrete changes to the maximum or minimum intensity. In response to increment and decrement, the intensity discretely increases or decreases by 1%. Continuous changes are described by illumination scenarios. A scenario defines a finite linear intensity change over time. The according input actScenario defines if there is an active scenario and which scenario it is. All inputs are activated per zone, i.e. there are copies of the a.m. inputs for each zone. Additionally, the user can trigger a scenario for all zones in parallel.

The reactions triggered by inputs bright, off, and actScenario are prioritized so that they possibly interrupt an active scenario. The current scenario is discarded and will never be continued. increment and decrement are only effective when no scenario is running.

[1]The CHARON model file is available at http://www.informatik.uni-bremen.de/agbs/research/hybriduml/hybrid-uml-profile-for-uml-2-0/

Illumination Scenarios

The illumination function defines a fixed number of scenarios that consist of the following parameters: STOP_INTENSITY is the target value – the scenario terminates when the intensity reaches this value. The parameter START_INTENSITY of the scenario is actually not used as start intensity, because in order to avoid discrete intensity changes the actual intensity is used as start value. In contrast, START_INTENSITY is used in conjunction with STOP_INTENSITY to determine if the scenario's slope is positive or negative. The slope's absolute value is defined by DURATION which is the time duration needed to change the intensity from the minimum to the maximum value (or vice versa). Finally, DELAY specifies a time duration that has to pass before the intensity change begins.

Illumination scenarios are related to locations in two ways. On the one hand, a scenario has different values of START_INTENSITY, STOP_INTENSITY, and DELAY for each cabin column. Note that DURATION is the same for all columns. On the other hand, a mapping between scenarios and cabin zones restricts the use of each scenario to certain cabin zones. This is intended to define the affected zones in case of parallel scenario activation.

HybridUML Model of the Cabin Illumination Function

For the case study, a cabin configuration with two cabin zones and two cabin columns is considered, so that four different locations have to be controlled independently. The complete cabin illumination function is represented by a composite agent Illumination containing four corresponding real-valued variables intensity $\in [0, 1]$.

The composite agent consists of two kinds of primitive agents – ScenarioControllers and AdjustmentControllers – that are organized as follows: there is one AdjustmentController per location which is responsible for discrete intensity changes, and there are as many ScenarioControllers for each location as enabled scenarios exist for the respective zone (see Fig. 8 and Fig. 9). Each ScenarioController thus represents the continuous intensity evolution in a location according to a specific scenario. Locations within zones that are not affected by a certain scenario (according to the scenario-zone-mapping) have no corresponding controller. As a result, a location has $0 \leq n_{loc} \leq scenarioCount$ ScenarioControllers. In this case study, $scenarioCount = 2$ scenarios are used.

In order to enhance consistency of the design, we add some constraint to the Illumination agent, capturing some requirements that clarify the dependencies between the parameters of the subagents. As the present HybridUML profile is based solely on the CHARON language, it currently does not feature this extension for agents. For the example of this section, this additional element of UML is used to improve the understanding of the illumination controller's structure.

Every set of controllers (of both kinds) with the same location modify the intensity of the location concurrently, therefore the intensity of each controller is mapped to the appropriate intensity of Illumination.

The input actScenario is an integer-valued variable that is expected to carry the number of the actual scenario in the respective cabin zone ($0 <$ actScenario $\leq scenarioCount$) or 0 denoting that no scenario is active. For reasons described below, actScenario has to be writable by the controller agents. Since the modification of this variable by one controller should not affect its value as seen by controllers associated with different locations (of the same zone), a copy exists for each location. This copy mechanism is not modeled but is a requirement to the environment in this case study. Furthermore, in order to activate a scenario in parallel for all zones, the environment has to set all actScenario variables simultaneously. Similar to intensity, all controllers of the same location share actScenario – but they coordinate on actScenario in a way that it is modified by at most one agent at a time.

The remaining inputs are modeled as discrete boolean variables. This is due to the underlying CHARON semantics that does not provide events. Similarly to actScenario, increment and decrement have to be writable for each AdjustmentController and have to be copied in order to be modified independently. In contrast, bright and off are not modified by any agent, therefore one variable per zone is sufficient. The boolean inputs only affect AdjustmentController because they trigger discrete illumination changes. Resulting interrupts of continuous changes are controlled via actScenario.

The behavior of each primitive ScenarioController agent is defined by its top-level mode ScenarioControl which is shown in Fig. 10. The valuation of actScenario directly determines which of the modes Idle and Scenario is active. This is enforced by their invariants in conjunction with the transitions between both modes. Scenario represents the situation where the scenario is active, Idle just waits for activation. Two different kinds of scenario termination are modeled here – interrupted termination caused by a change of actScenario, and regular termination after the target value is reached by intensity. Here, actScenario has to be writable because the actual scenario is reset by the scenario controller after regular termination. Thus it is not exclusively controlled by the environment.

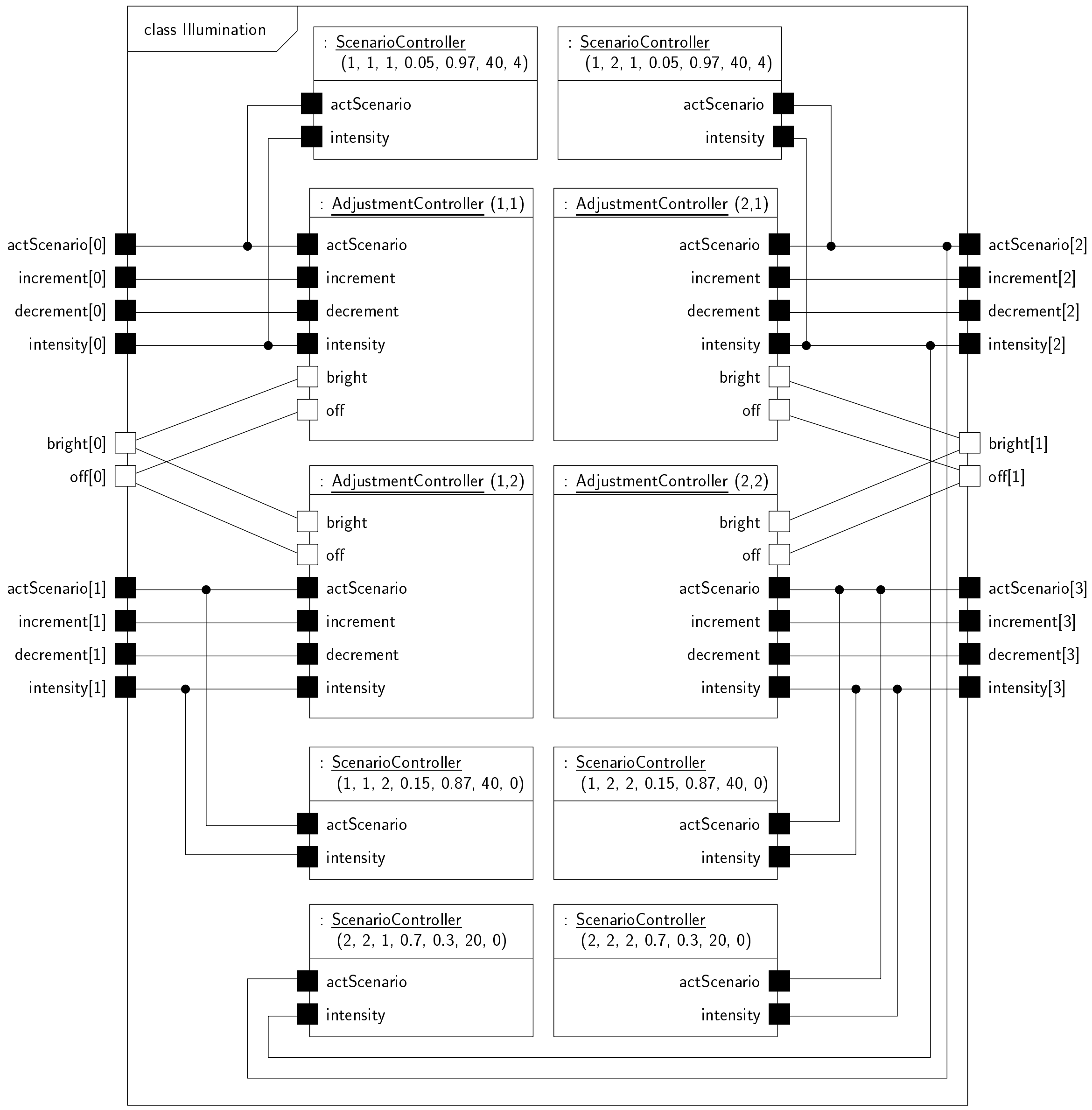

Figure 8: Composite Structure Diagram View of Composite Agent Illumination.

The mode Scenario (see Fig. 11) realizes the linear illumination intensity change for the controller's location according to the values of the parameters START_INTENSITY, STOP_INTENSITY, DELAY, and DURATION, taking advantage of HybridUML's discrete-continuous specification formalism – discrete control flow defined by transitions integrates with continuous valuation flows within leaf modes.

Scenario contains two submodes – Wait and Run. Initially, Wait is entered and remains active for DELAY time units. The agent's clock variable delayClock is used to model this. After exiting Wait, slope is calculated correspondingly to the values of START_INTENSITY, STOP_INTENSITY, and DURATION. Then, submode Run continuously changes intensity as long as the target value STOP_INTENSITY is not reached yet. As soon as the target value is reached, the right transition of Fig. 11 originating at Run is taken and intensity is set to STOP_INTENSITY. This is necessary, because there are cases in which STOP_INTENSITY will never be reached – then the transition is taken immediately and intensity is updated discretely. This is possible, because the CHARON semantics allows the traversal of modes even if their invariants are violated, provided that no time passes.

Besides some integrity constraints, the invariant of Scenario ensures that a change of the global variable actScenario interrupts the mode. But due to a specific restriction of the CHARON semantics, a group transition cannot be used to interrupt this mode: if a mode is exited via its default exit, then it has to be re-entered via its default entry – therefore the history feature is mandatory for a mode after being interrupted by a group

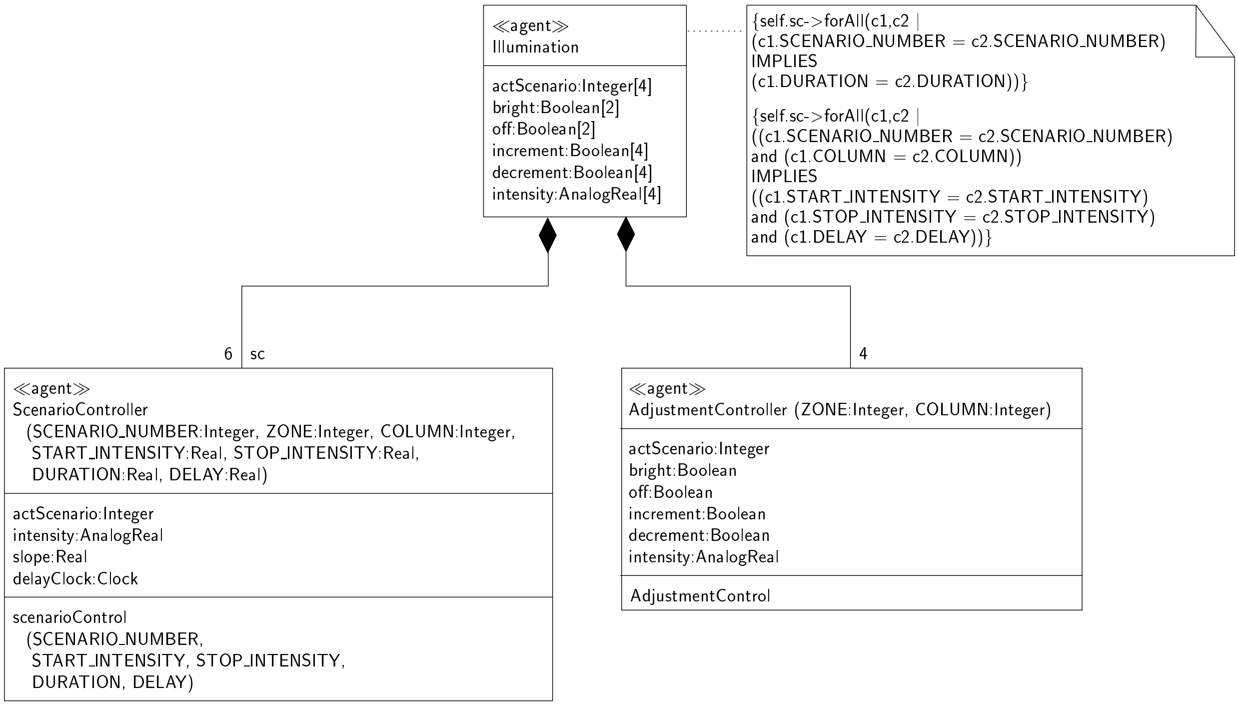

Figure 9: Class Diagram View of Composite Agent Illumination and Primitive Agents ScenarioController and AdjustmentController.

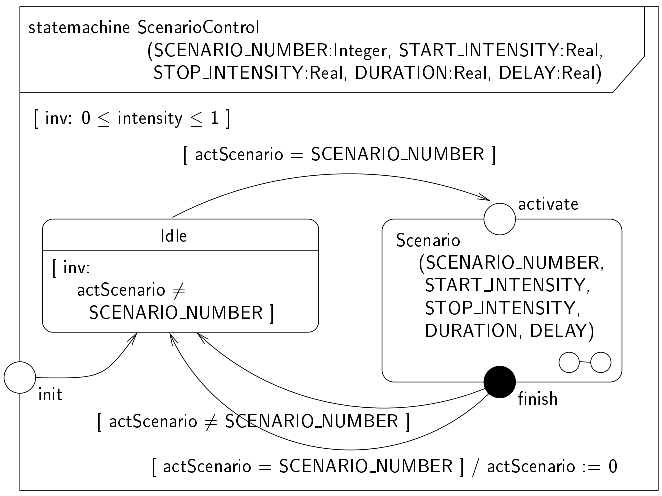

Figure 10: Top-level Mode Defining the Behavior of one ScenarioController.

transition. Since any illumination scenario must be re-initialized, i.e. since it must not use the history when it is re-entered, the interrupt is modeled explicitly by extending the guard of the outgoing transition from Wait and by adding a further transition from Run to finish. The a.m. traversal of modes is utilized here, too.

The top-level mode AdjustmentControl that is shown in Fig. 12 defines the behavior of the primitive agent AdjustmentController.

If one of the inputs bright or off is active, the submode Idle is active, otherwise AdjustOrWait has control. This is enforced by the invariants of both modes (see also Fig. 13). The activation of one of these inputs (assuming that both are inactive) triggers the corresponding transition and sets intensity accordingly. Also it is ensured that afterwards no scenario is active, possibly interrupting a currently active scenario indirectly – setting actScenario= 0 enforces all ScenarioControllers of this location to deactivate their continuous adjustments of intensity.

As soon as both bright and off are inactive, the submode AdjustOrWait (see Fig. 13) is (re-)entered. It contains two submodes. Adjust represents the possibility for the stepwise intensity adjustment using increment and

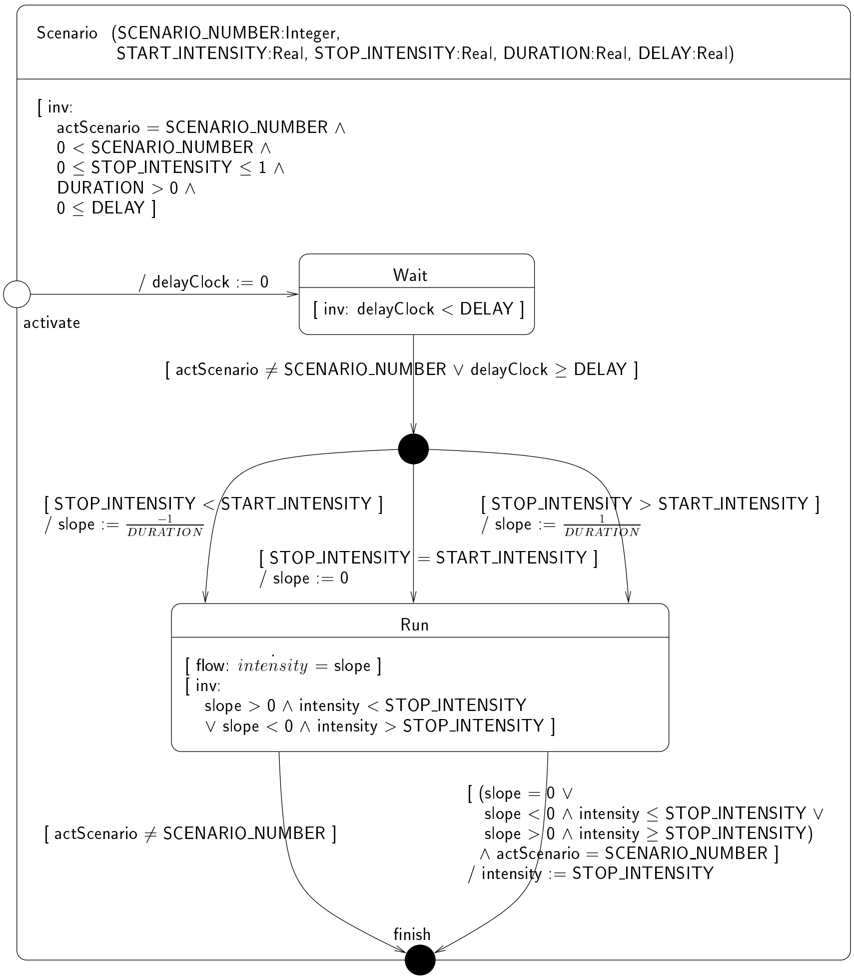

Figure 11: Submode Specifying the Behavior in Case that a Specific Scenario is Active.

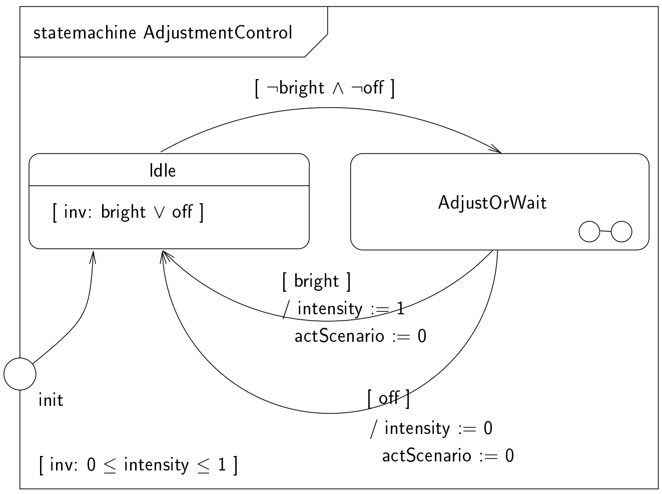

Figure 12: Top-level Mode Defining the Behavior of one AdjustmentController.

decrement. Attached to this mode are four self-transitions that implement the actual adjustments, considering the boundaries given by the maximum and the minimum intensity. Each transition also resets its corresponding input so that it is taken exactly once.[2]

Submode WaitForScenario in contrast represents the inability for stepwise adjustment while a scenario is

[2] This is the workaround for the missing event concept that would have been more suitable here.

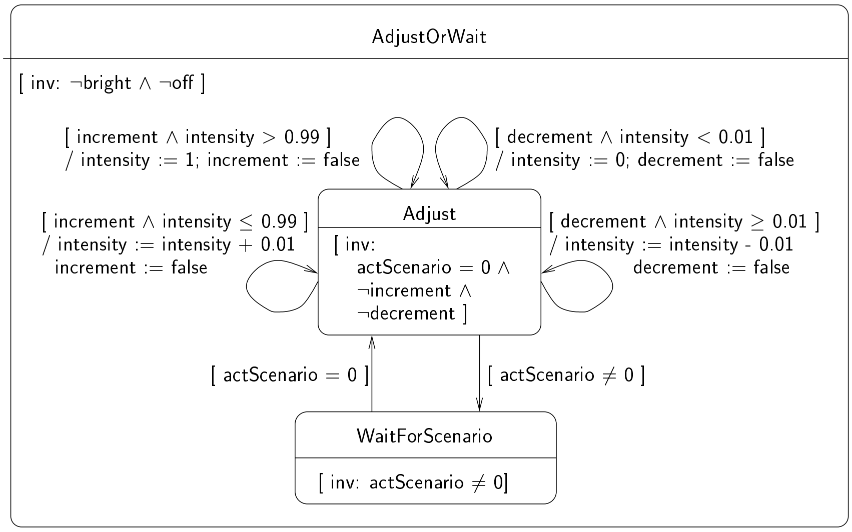

Figure 13: Submode Specifying the Behavior of AdjustmentController in Case that bright and off are Inactive – Stepwise Adjustment or Waiting for Scenarios to Finish.

running. Switching between Adjust and WaitForScenario depends on the value of actScenario that may be altered by the scenario controller agents, in particular.

6 Discussion

Using the profile construction mechanism, we have introduced an extension of UML 2.0 for hybrid systems modeling. The profile adopts the CHARON semantics.

Benefits resulting from the profile construction. We have followed the structuring approach suggested by other authors for various profiles defined in the context of UML 1.4 [OMG02b, OMG02a] and UML 2.0 [EIF+03]. We consider the firmly structured description plan as a major benefit: The mapping of original CHARON semantic definition fragments to the appropriate profile description sections was quite straight-forward and resulted in a well-defined standardized semantics description structure which may be easier to understand than the intuitive structure used in the original publications [AGLS01, ADE+01, ADE+03]. From these experiences we suggest that this structure for profile construction should be adopted as a guideline within the UML 2.0 standard.

Furthermore, the richness of the UML syntax results in a more elegant syntactic appearance than the original CHARON representation. For example, the distinction between states and pseudostates available in UML syntax is very helpful to distinguish between locations where the system may reside in for a positive amount of time and locations the system just passes through in the process of decision making. Note that CHARON does not provide this distinguishing feature on syntax level, but the possibility to define states which are immediately left after entry (because the state invariant evaluates to *false*) provides the same semantic expressiveness.

The general UML approach which does not fix semantic meaning in the mathematical sense seems to be appropriate from our point of view: It appears to be quite natural that the large number of syntactic constructs meant to satisfy every possible modeling situation results in different interpretations, depending on the specific system to be modeled. For example, the mathematical statecharts semantics given in [DJHP98] assigns a well-defined meaning for statecharts interpretations in the context of discrete-time control systems. This is suitable for model checking of assertions and development of controllers processing discrete data. However, there is no straight-forward extension of this semantics to hybrid systems, since the latter are based on a dense-time model. As a consequence, a fixed semantic statechart model like [DJHP98] would complicate or even prevent the UML extension presented here.

Finally, being able to write hybrid specifications with any CASE tool supporting UML 2.0 and the profiling mechanism is a considerable advantage, protecting the investments into new CASE tools and their company-wide introduction. In contrast to this, non-standard UML customizations or extensions as, for example, the real-time extension [Sel98], introduce additional syntactic constructs in an uncontrolled way, resulting in a UML "dialect" only supported by a small number of CASE tools.

Feedback from UML to CHARON. Analyzing the case study and UML language constructs available, it is possible to get a feedback about possible improvements of CHARON: CHARON requires that modes which have been left via the default exit may only be re-entered via default entries. The closure operation implies that this always acts like a deep history. This is a severe restriction; in many modeling situations application of the more flexible history operators of UML statecharts are desirable. An analysis of the CHARON semantics shows that this extension could be made without fundamental changes of the existing CHARON interpretation.

In contrast to this, the extension to mixed shared variable/event interfaces leads to a more complex semantics, as can be seen in the parallel composition introduced for Hybrid Automata [Hen96]. Furthermore, we advocate to keep the CHARON restriction not to allow parallel sub-modes, though it leads to less elegant specifications: It is well known that the semantic complexity is dramatically increased when switching from networks of sequential cooperating components to systems where arbitrary modules may contain parallel sub-components.

Critical remarks about graphical languages in general. The small case study introduced above already shows the fundamental problems of graphical specification languages in general, which have not (and probably could not) been solved by UML or any other graphical formalism: The first problem is how to deal with large numbers of components, such as the agents ranging over scenarios, zones, columns introduced in the top-level agent specification of the case study. To our knowledge no graphical formalism allows to describe large collections of parameterized components adequately on a graphical level. Here the algebraic expressions like parameterized parallel composition of processes in CSP [Ros97] are much more elegant, though they cannot transport intuitive meaning by drawn boxes, arrows, etc.

The second problem consists in the fact that invariants, flow conditions, assignments, parameterized method calls, and other items still have to be expressed on textual level. As a consequence, graphical specifications like the statecharts described in the case study above tend to become overloaded with text, as soon as conditions and actions become more complex. Even if we can separate complex expressions from the graphical specification by using identifiers pointing to separate textual specifications, the more complex specification aspects tend to be moved to the textual level, so that the graphical level frequently expresses just the more trivial specification aspects.

Acknowledgments. 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* [DFG].

References

[ACH+95] R. Alur, C. Courcoubetis, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995. A preliminary version appeared in the proceedings of 11th. International Conference on Analysis and Optimization of Systems: Discrete Event Systems (LNCI 199).

[ADE+01] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling of embedded systems. *Lecture Notes in Computer Science*, 2211:14–31, 2001.

[ADE+03] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, January 2003.

[AGLS01] Rajeev Alur, Radu Grosu, Insup Lee, and Oleg Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 33–48, 2001.

[Ber03] Kirsten Berkenkötter. Using UML 2.0 in real-time development - a critical review. SVERTS Workshop at the ⟨⟨ UML ⟩⟩ 2003 Conference, October 2003. http://www-verimag.imag.fr/EVENTS/2003/SVERTS/.

[CHA] CHARON toolkit and information. http://www.cis.upenn.edu/mobies/charon/.

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

[DJHP98] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. *Lecture Notes in Computer Science*, 1536:186–238, 1998.

[EIF+03] Ericsson, IBM, FOKUS, Motorola, Rational, Softeam, and Telelogic. UML Testing Profile (Draft Adopted Specification), July 2003.

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

[HP95] Ute Hamer and Jan Peleska. Z Applied to the A330/340 CIDS Cabin Communication System. In Michael Hinchey and Jonathan Bowen, editors, *Applications of Formal Methods*, pages 253–284, Englewood Cliffs NJ, 1995. Prentice Hall International.

[JRB99] Ivar Jacobson James Rumbaugh and Grady Booch. *The Unified Modeling Language – Reference Manual*. Addison-Wesley, 1999.

[OMG02a] OMG. UML Profile for Schedulability, Performance, and Time Specification. http://www.omg.org/cgi-bin/doc?ptc/2003-03-02, March 2002.

[OMG02b] OMG. Unified Modeling Language Specification (Action Semantics), January 2002.

[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.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall International, Englewood Cliffs NJ, 1997.

[Sel98] Bran Selic. Using UML for Modeling Complex Real-Time Systems. In *Languages, Compilers, and Tools for Embedded Systems, ACM SIGPLAN Workshop LCTES'98, Montreal, Canada, June 1998, Proceedings*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–260. Springer, 1998.

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

A A HybridUML Profile based on CHARON

HybridUML is a UML 2.0 profile that is based on CHARON. It takes a subset of UML, modifies it according to CHARON and gives it precise semantics. Therefore the most important constraint on applying the HybridUML profile is using only the model elements specified in it. Other elements are not covered in the profile including sequence diagrams, activity diagrams, and so on.

A.1 PrimitiveType

A.1.1 Real

CHARON uses typed datatypes. In contrast to Integer, String, and Boolean, real numbers are not covered in UML. We extend PrimitiveType for getting a Real datatype.

Description

Real is an instance of PrimitiveType (see Fig. 14). It defines a real-valued number.

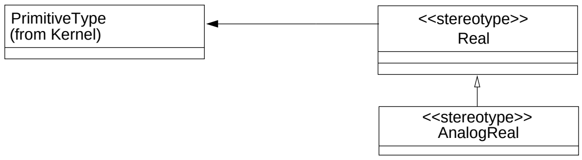

Figure 14: Stereotype for Supporting Real-Valued Numbers

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

No additional constraints.

Semantics

Real instances will be values of $\mathbb{R}$. If used for model checking, real instances will be values of $\mathbb{Q}$.

Notation

Real will appear as the type of attributes (see Fig. 15). Real instances will be values of $\mathbb{R}$ associated to slots.

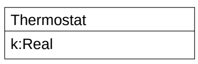

Figure 15: Notation of Real

A.1.2 AnalogReal

We also need analog real numbers, i.e. real numbers whose value is varying with passing time. The way the value is changing is given in an expression of the type RTExpression that is described in the next chapter. AnalogReal variables and RTExpressions are attached to one another in Modes.

Description

AnalogReal is a generalization of Real (see Fig. 14). It defines an analog real number. AnalogReal numbers are used in statecharts for describing flow conditions and invariants.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

No additional constraints.

Semantics

AnalogReal instances will be values of $\mathbb{R}$. The value is changing over time according to a given RTExpression in the context of a Mode.

Notation

AnalogReal will appear as the type of attributes (see Fig. 16). AnalogReal instances will be values of $\mathbb{R}$ associated to slots. The RTExpression is given in Modes where the AnalogReal numbers are used.

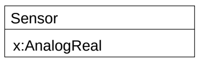

Figure 16: Notation of AnalogReal

A.2 Expression

A.2.1 RTExpression

For describing the evaluation of AnalogReal variables, different expressions are needed, i.e. differential expressions and algebraic expressions. Invariant expressions are needed for defining state invariants.

Description

RTExpression is an instance of Expression (see Fig. 17). It defines mathematical an logical terms that may be dependent on time. RTExpression is an abstract metaclass that cannot be instantiated.

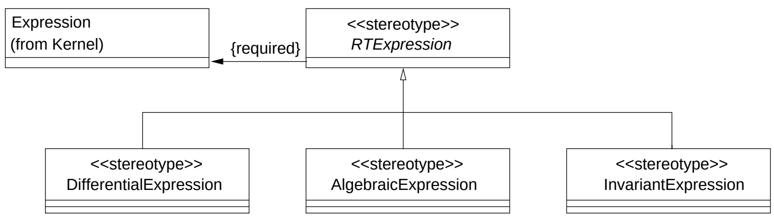

Figure 17: Stereotypes for RTExpressions

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- The real-time expression is given in the attribute symbol as a string, just as in Expression. Furthermore, the expression must be mathematically or logically evaluable.

Semantics

The semantics are given by the concrete subtypes.

Notation

Notation is given by the concrete subtypes.

A.2.2 AlgebraicExpression

Description

AlgebraicExpression is an generalization of RTExpression (see Fig. 17). It describes algebraic terms independent on time.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- Attribute *symbol* must give a mathematical, non-differential term. All variable names in *symbol* must correspond to variables in the model:
  Let V be the set of variables in the Mode owning the RTExpression and V_{num} the set of variables of type Real, AnalogReal or Integer. Then $V_{num} \subseteq V$. If the set of variables in *symbol* is V_s, then $V_s \subseteq V_{num}$.

Semantics

An AlgebraicExpression is a function $f : \mathbb{R}^n \to \mathbb{R}$

Notation

Mathematical term, non-differential, e.g. $x = f(y, z)$.

A.2.3 DifferentialExpression

Description

DifferentialExpression is a generalization of RTExpression (see Fig. 17). It describes differential terms dependent on time.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- Attribute *symbol* must give a differential equation dependent on time. All variable names in *symbol* must correspond to variables in the model:
  Let V be the set of variables in the Mode owning the RTExpression and V_{num} the set of variables of type AnalogReal, Real, or Integer. Then $V_{num} \subseteq V$. Let V_s be the set of variables in *symbol*. Then $V_s \subseteq V_{num}$.

Semantics

A DifferentialExpression is a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$

Notation

Mathematical term, e.g. $\dot{x} = f(x, u)$, where $\dot{x}\ is\ \frac{dx}{dt}$.

A.2.4 InvariantExpression

Description

InvariantExpression is a generalization of RTExpression (see Fig. 17). It is used for modeling invariants a variable must fulfill. This is useful in Modes.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- Attribute *symbol* must give an invariant as logical expression. All free variable names in *symbol* must correspond to variables in the model:
  Let V be the set of variables in the Mode owning the RTExpression and V_s the set of free variables in $symbol$. Then $V_s \subseteq V$.

Semantics

An InvariantExpression is an invariant given as logical expression.

Notation

Logical expression, e.g. $x \leq c$.

A.3 Constraint

For describing the flow of analog variables in CHARON, constraints are used. These are given in Modes that use the corresponding variable.

A.3.1 RTConstraint

RTConstraint is a UML constraint that is restricted in order to describe an RTExpression. Therefore, DifferentialExpressions, AlgebraicExpressions, and InvariantExpressions can be used.

Description

RTConstraint is an instance of Constraint. It holds an RTExpression. DifferentialExpressions describe AnalogReal variables, AlgebraicExpressions Real and Integer variables. InvariantExpressions are used with all variable types.

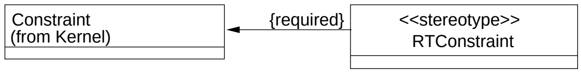

Figure 18: Stereotype for RTConstraint

Attributes

No additional attributes.

Associations

No additional associations

Constraints

- The given specification must be an RTExpression, i.e.
  self.specification->forAll(oclIsKindOf(RTExpression))
- There must be exactly one specification given, i.e.
  self.specification->size = 1

Semantics

According to the included RTExpression.

Notation

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 (see Fig. 28).

A.4 Time

A.4.1 Clock

For modeling time, we need clocks. This is done by using a variable of type AnalogReal that uses a differential equation for modeling the flow of time.
Therefore we inherit from AnalogReal to get a clock. We do not use the UML 2.0 time model as it has no formal semantics and it is not powerful enough for our purposes.

Description

A Clock is modeled by an AnalogReal variable (see Fig. 19). The flow of time is specified as a DifferentialEquation.

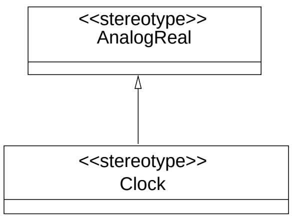

Figure 19: Stereotype for Clock

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

Let t be the value of a Clock instance. Then the following expression always holds: $\dot{t} = 1$

Semantics

Time flow is expressed by DifferentialExpression $\dot{t} = 1$ if t is the value of a Clock instance.

Notation

Clocks are modeled as attribute of type Clock, e.g. x:Clock. As the differential equation is explicitly given, it is not added as a constraint following the variable (see Fig. 21).

A.5 Structure

A.5.1 Agent

Agents are the main element for modeling structure with CHARON. They consist of a set of variables, a set of Modes, i.e. statecharts, and a set of initial states. Agents can be composed of other Agents, respectively Agent instances.

In HybridUML, Agents are stereotypes of classes. They consist of VariablePorts, 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).

Description

An Agent is an instance of Class (see Fig. 20) that can own an internal structure (see Fig. 22). The internal structure consists of Agent instances. Agents communicate by VariablePorts and VariableConnectors.

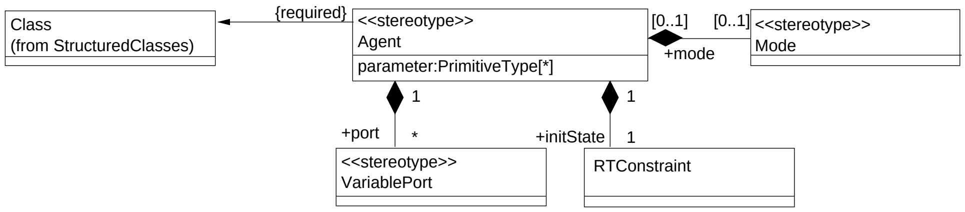

Figure 20: Stereotype for Agent

Attributes

parameter:PrimitiveType[*]

Associations

- port:VariablePort[*]
- mode:Mode[0..1]
- initState:RTConstraint[1]

Constraints

- Agents do not hold operations, i.e.
  self.ownedOperation->size = 0
- All parts of the internal structure are Agent instances, i.e.
  self.part->forAll(oclIsTypeOf(Agent))
- All connectors are VariableConnectors, i.e.
  self.ownedConnector->forAll(oclIsTypeOf(VariableConnector))
- The expression given in initState must correspond to the variables of the interface-typed ports:
  Let V_i be the set of variables in initState and V the set of variables of Agent A. Then $V_i \subseteq V$.
- initState cannot be a differential term, i.e.
  not (self.initState.specification.oclIsTypeOf(DifferentialExpression))
- If there is an internal structure, i.e. the Agent is a composite Agent, it has no own mode but the modes of the Subagents:
  self.part->size > 0 implies self.mode->size = 0
- Each Agent that owns a Mode is a Thread, i.e.
  self.mode->size = 1 implies self.isActive = true

- Modes are constant, i.e. they cannot be changed by agent instances:
  self.mode->forAll(isReadOnly = true)

- Parameters are constants, i.e.
  self.parameter->forAll(isReadOnly = true)

Semantics

An Agent $< TM, V, I >$ consists of a set of variables V, a set of inititial states I, and a set of top-level Modes TM. We distinguish *primitive* 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. The set of variables V is partitioned into local variables V_l and global variables V_g. The VariablePorts are connected to required or provided VariableInterfaces which include one variable each. The set V_P of all port variables together with the set V_C of all variables of type Clock correspond to the set of global variables V_g as defined in CHARON. Note that Clocks are global for all parts of an Agent. We do not model them as VariablePorts as this would be obfuscating.

Local variables correspond to the VariablePorts owned by a Subagent together with the parameters V_{Pa}. Therefore V_l is the set of variables owned by their corresponding VariableInterfaces V_I and V_{Pa}. Parameters are constant local variables for usage in constraints of all kind.

The top-level Modes TM define the behavior of the system. A primitive Agent has one (its own) top-level Mode, a composite Agent has as many top-level Modes as it has Subagents. Vice versa, each Mode is either a top-level Mode connected to an Agent, or a Submode and therefore implicitly connected to an Agent. It can see only the variables of its own Agent, i.e. denoting the variables of the top-level Modes by $TM.V$, we have $TM.V \subseteq V$. All VariablePorts of an Agent are connected to its Mode, i.e. $V_g \subseteq TM.V_g$. A valuation for V is a function mapping variables to their (type-correct) values. The set of valuations over V is denoted Q_V. The set $I \subseteq Q_V$ of initial states specifies possible initializations of the variables of the Agent and is specified by the collection of all initStates given in the Agent instances.

The semantics of an Agent are defined by the (trace) behavior of its top-level Modes, constructed from the respective relations R^C and R^D which are explained in detail in Section A.6.1. A trace of a Mode M is a projection of an execution of that Mode onto its global variables; it is described as a sequence $s_0 \xrightarrow{\lambda_1} s_1 \xrightarrow{\lambda_2} s_2 \dots \xrightarrow{\lambda_n} s_n$, with $s_i \in Q_{M.V_g}$ a data state, i.e., a valuation of the (global) variables of M. Here, λ_i ranges over $\mathcal{F}_{M.V_g} \cup \{o, \varepsilon\}$, where o denotes a discrete step of that Mode, ε denotes an environment step w.r.t. M, and $\lambda_i \in \mathcal{F}_{M.V_g}$ represents a *flow* for the variable set $M.V_g$, i.e., a differentiable function from a closed interval of non-negative reals $[0, \delta]$ to Q_{V_g} (we refer to δ as the duration of that flow).

An execution of an Agent A follows a trajectory, which starts in one of the initial states and is a sequence of flows interleaved with discrete updates to the variables of the Agent. For a fixed initial state s_0, each Mode $M \in TM$ starts out in a configuration $(init_M, s_M)$, where $init_M$ is the distinguished non-default entry point of M and $S_0[M.V] = s_M$. Note that as long as there is a Mode M whose control state is at $init_M$, no continuous steps are possible.

The choice of a continuous step involving all Modes or a discrete step in one of the Modes is left to the environment. Before each discrete step, there is an environment step, which takes the control point of the chosen Mode from dx to de and leaves all the private variables of all top-level Modes intact. After that, a discrete step of the chosen Mode happens, bringing control back to dx, where again some continuous step is possible. Note that these are the only steps possible for a top-level Mode, as his set of control points contains exactly $init_M$, de and dx.

Thus, an execution of A with $\mid TM \mid = k$ is a sequence $s_0 \xrightarrow{o} s_1 \xrightarrow{o} \dots s_k \xrightarrow{\lambda_1} s_{k+1} \xrightarrow{\lambda_2} \dots$ such that

- for every $0 \leq i < k$, there is $M \in TM$ such that $(s_i[M.V], s_{i+1}[M.V]) \in M.R^D_{init_M, dx}$. That is, the first k steps initialize the top-level Modes of A.

- for every $i \geq k$, one of the following holds:
  - $s_i \xrightarrow{f} s_{i+1}$ such that f is defined on $[0, t]$ and $f(t) = s_{i+1}$, and for every Mode $M \in TM$, $(s_i[M.V], f[M.V]) \in M.R^C$; that is, the step is a continuous step, in which every Mode takes part;
  - $s_i \xrightarrow{\varepsilon} s_{i+1}$ such that for every Mode $M \in TM$, $s_i[M.V_l] = s_{i+1}[M.V_l]$; that is the step is an environment step;
  - $s_i \xrightarrow{o} s_{i+1}$ with $i > k$, there is $M \in TM$ such that $(s_i[M.V], s_{i+1}[M.V]) \in M.R^D_{de,dx}$; that is, the step is a discrete step by one of the Modes.

Note that environment steps in Agents and in Modes are different. In an Agent, an environment step may contain only discrete steps, since all Agents participate in every continuous step. The environment of a Mode can engage in a number of continuous steps while the Mode is inactive.

A trace of an Agent A is an execution of A, projected onto the set of its global variables. The denotational semantics of an Agent consists of its set of global variables and its set of traces.

The composition of two Agents $A_1 \| A_2$ is an Agent $A =< TM, V, I >$ defined as follows: $A.TM = A_1.TM \cup A_2.TM$, $A.V_g = A_1.V_g \cup A_2.V_g$, $A.V_l = A_1.V_l \cup A_2.V_l$, and if $s \in A.I$ then $s[A_1.V] \in A_1.I$ and $s[A_2.V] \in A_2.I$.

Given an Agent $A =< TM, V, I >$, the hiding operator makes a set of variables $V_h \subseteq V$ private, i.e., $A \setminus \{V_h\} = \ < TM, V', I >$, with $V_l' = V_l \cup V_h$ and $V_g' = V_g - V_h$. The trace of A, projected onto the set of global variables of $A \setminus \{V_h\}$ (i.e. $V_g - V_h$), is a trace of $A \setminus \{V_h\}$. These operations can be reflected already in the representation of an Agent's internal structure in a composite structure diagram.

Notation

Agents are depicted like UML classes with internal structure. In a class diagram, the internal structure is visualized as aggregated classes (see Fig. 21). The parameter list of each Agent is given behind its name in parentheses in the first compartment of the class symbol.

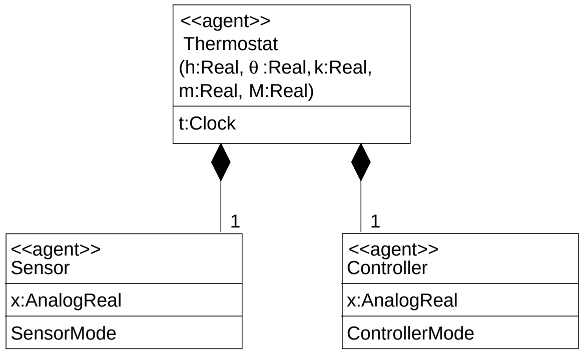

Figure 21: Notation for Agent in Class Diagram

VariablePorts and their included VariableInterfaces and variables are given as attributes in the second compartment of the class symbol. In the class diagram, only the name and type of the included variable is given.

Optionally, in the third compartment of the class symbol 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. The name of the Agent is given in the upper left corner with the keywort *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 (see Fig. 22).

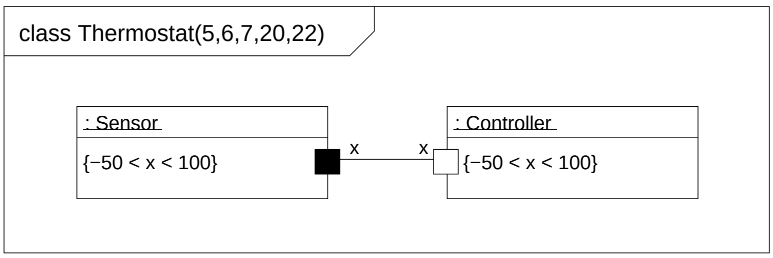

Figure 22: Notation for Agent in Composite Structure Diagram

Agent instances are visualized as objects in composite structure diagrams (see figure 22). 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, 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 (see Fig. 28).

A.5.2 VariableConnector

UML 2.0 ports are linked by connectors, VariablePorts are linked by VariableConnectors. VariableConnectors link VariablePorts with same (required or provided) VariableInterfaces.

Description

VariableConnectors are instances of Connector (see Fig. 23). They link VariablePorts whose required or provided interfaces must be instances of the same VariableInterface, i.e. they mirror the same global variable.

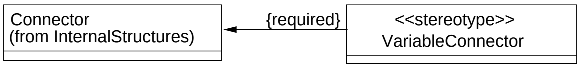

Figure 23: Stereotype for VariableConnector

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- All ConnectorEnds are VariableConnectorEnds, i.e.
  self.end->forAll(oclIsTypeOf(VariableConnectorEnd))
- All VariableConnectorEnds mirror the same VariableInterface, i.e. the same global variable:
  self->forAll(e1, e2: VariableConnectorEnd |
  if (e1.role->exists(provided)) then
  def : value1:PrimitiveType = e1.role.provided.ownedAttribute
  else
  def : value1:PrimitiveType = e1.role.required.ownedAttribute
  endif
  and if(e2.role->exists(provided)) then
  def : value2:PrimitiveType = e2.role.provided.ownedAttribute
  else
  def : value2:PrimitiveType = e2.role.required.ownedAttribute
  endif
  and value1 = value2
  and if(value1.oclIsTypeOf(Real)) then value2.oclIsTypeOf(Real) endif
  and if(value1.oclIsTypeOf(AnalogReal)) then value2.oclIsTypeOf(AnalogReal) endif
  and if(value1.oclIsTypeOf(Integer)) then value2.oclIsTypeOf(Integer) endif
  and if(value1.oclIsTypeOf(UnlimitedNatural)) then value2.oclIsTypeOf(UnlimitedNatural) endif
  and if(value1.oclIsTypeOf(Boolean)) then value2.oclIsTypeOf(Boolean) endif
  and if(value1.oclIsTypeOF(String)) then value2.oclIsTypeOf(String) endif)

Semantics

A VariableConnector connects VariablePorts. All ends of the connector mirror the same variable. This variable is defined by the VariableInterface owned by a VariablePort.

Notation

VariableConnectors are solid lines between ports (see Fig. 22).

A.5.3 VariableConnectorEnd

Description

VariableConnectorEnds are instances of ConnectorEnd (see Fig. 24). They are the ends of VariableConnectors. They are always attached to VariablePorts.

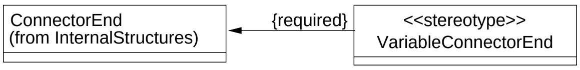

Figure 24: Stereotype for VariableConnectorEnd

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- All VariableConnectorEnds are attached to VariablePorts, i.e.
  self.role->forAll(oclIsTypeOf(VariablePort))

Semantics

VariableConnectorEnds are the ends of VariableConnectors. They are always linked to VariablePorts.

Notation

VariableConnectorEnds do not have a notation.

A.5.4 VariableInterface

In CHARON, Agents talk to each other exclusively over global variables. They are visualized in the same way as UML 2.0 Ports.

We therefore need a VariableInterface that mirrors exactly one variable. In combination with VariablePorts and VariableConnectors, the concept of global variables as used in CHARON can be modeled.

Description

VariableInterface is an instance of Interface (see Fig. 25). A VariableInterface is associated to one global variable that must be of type PrimitiveType.

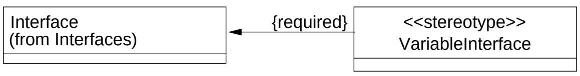

Figure 25: Stereotype for VariableInterface

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- VariableInterfaces own exactly one property, i.e.
  self.ownedAttribute->size = 1
- The property owned by VariableInterface is of type PrimitiveType, i.e.
  self.ownedAttribute.oclIsKindOf(PrimitiveType)
- VariableInterfaces do not own operations, i.e.
  self.ownedOperation->size = 0
- VariableInterfaces are not nested, i.e.
  self.nestedInterface->size = 0
- VariableInterfaces will not be redefined, i.e.
  self.redefinedInterface->size = 0

Semantics

VariableInterfaces are used by VariablePorts in HybridUML. They represent global variables. Each interface owns a property that mirrors the value of a global variable, i.e. the properties of connected interfaces must have the same value. Concrete semantics for this is given in VariableConnector.

Notation

VariableInterfaces own exactly one property, its name is given as the name of the VariableInterface, respectively as the name of the VariablePort that owns the interface. VariableInterfaces are only visualized in combination with ports (see Fig. 22).

A.5.5 VariablePort

Ports serve as interaction points in UML 2.0. They own required and provided interfaces. We therefore introduce VariablePorts that own required and provided VariableInterfaces.

Required VariableInterfaces correspond to read access and provided VariableInterfaces correspond to write access with respect to CHARON. Write access means read/write access.

Description

A VariablePort is an instance of Port (see Fig. 26). It has only required and provided VariableInterfaces. VariablePorts are always service ports and behavior ports, i.e. they are interaction points of Agents and connected to a state machine called Mode. Each VariablePort owns exactly one VariableInterface.

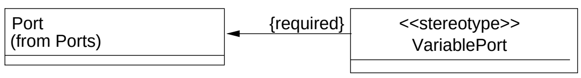

Figure 26: Stereotype for VariablePort

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- VariablePorts are service ports, i.e. they specify the functionality of the Agent they belong to:
  self.isService = true
- VariablePorts are behavior ports, i.e. they are connected to the Mode of the Agent they belong to:
  self.isBehavior = true
- VariablePorts own only VariableInterfaces, i.e.
  self.required->forAll(oclIsTypeOf(VariableInterface)) and
  self.provided->forAll(oclIsTypeOf(VariableInterface))
- VariablePorts own exactly one required or provided VariableInterface, i.e.
  self->exists(required) implies (self.required->size = 1 and self.provided->size = 0)
  and self->exists(provided) implies (self.provided->size = 1 and self.required->size = 0)
- VariablePorts own exactly one VariableInterface at all, i.e.
  if self->exists(required) then not(self->exists(provided)) endif and
  if self->exists(provided) then not(self->exists(required)) endif
- A required interface means read access, i.e.
  if (self->exists(required)) then
  post: self.required.ownedAttribute = self.required.ownedAttribute@pre
- A provided interface means read/write access, i.e. normal behavior.

Semantics

VariablePorts are used in connection with VariableInterfaces. They are access points for Agents. VariablePorts are connected by VariableConnectors.

Notation

VariablePorts are depicted like ports, i.e. a rectangle on the boundary of the owning classifier. Instead of visualizing the attached interfaces in lollipop-notation, a required interface is a white filled rectangle and a provided interface is a black filled rectangle (see Fig. 22). As every port is connected to a Mode, the state symbol that indicates behavior ports will be omitted. In class diagrams, only the variable owned by the VariableInterface of the port will be shown (see Fig. 21).

A.6 Behavior

A.6.1 Mode

In CHARON, Modes are used for describing the behavior of an Agent. These are hybrid state machines.

Description

Mode is an instance of StateMachine describing an Agent's behavior. 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. Every Mode has a default entry point *de* and a default exit point *dx*.

A Mode that is not contained by any other Mode is called top-level Mode. Each top-level Mode has a single non-default entry point called *init* point, and no non-default exit point. A Mode that is contained by another Mode is called Submode. A Mode without Submodes is called leaf Mode.

Top-level Modes are connected to an Agent. They use the global variables defined in this Agent. Analog variables are updated according to constraints while the state machine is in a Mode. Discrete variables are only updated when a transition is taken. Modes can have parameters for better scalability. Preemption is modeled by using the default exit point *dx* as source of a group transition.

Attributes

- /isTopLevelMode : Boolean
- parameter:PrimitiveType[*]

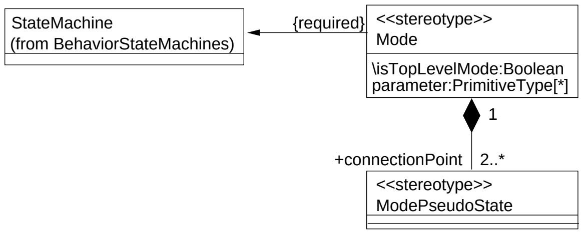

Figure 27: Stereotype for Mode

Associations

- connectionPoint : ModePseudostate[2..*]

Constraints

- Each Mode has exactly one region, i.e.
  self.region->size = 1
- Modes are not extended, i.e.
  self.extendedStateMachine->size = 0
- Modes will not be redefined, i.e.
  self.redefinitionContext->size = 0
- The context of a Mode is an Agent (if there is a context), i.e.
  self.context.oclIsTypeOf(Agent)
- Each Mode has exactly one default entry and exit point, i.e.
  self.connectionPoint->select(v | v.kind = defaultEntry)->size = 1
  self.connectionPoint->select(v | v.kind = defaultExit)->size = 1
- Each top-level mode has exactly one non-default entry point and no non-default exit point, i.e.
  self.isTopLevelMode=true implies
  (self->connectionPoint->select(v | v.kind = entryPoint)->size = 1 and
  not(self->exists(connectionPoint.kind = exitPoint))
- Parameters are constants, i.e.
  self.parameter->forAll(isReadOnly = true)
- Each Mode has a $default\ transition$ from de to dx, i.e.
  self.region.transitions->exists(target.kind = defaultExit and source.kind = defaultEntry
  and target.owner = self and source.owner = self)

Semantics

A mode M is a tuple $< E, X, V, SM, Cons, T >$, where V is the set of variables V_g that are owned by the corresponding primitive Agent, together with the parameters of that Mode which are constants used in constraints of all kinds in the Mode and its Submodes, and which can be interpreted as set of local variables V_l. V_a is the set of variables of type AnalogReal. SM is a set of Submodes, i.e. the set of ModeStates of the Mode. Each global variable of a Submode is also a variable of the parent Mode. That is, if variable $N \in SM$ then $N.V_g \subseteq V$.

E is a set of entry control points, i.e. ModePseudostates of kind defaultEntry and entryPoint. X is a set of exit control points, i.e. ModePseudostates of kind defaultExit and exitPoint. The set C of all control points is $C = E \cup X \cup SM.E \cup SM.X$

$Cons$ is a set of constraints, i.e. the set of the stateInvariant and the stateFlow of each ModeState. $Cons$ includes one *invariant* I of type InvariantExpression that defines when a ModeState is active. For each analog variable $x \in V_a$, $Cons$ may contain an *algebraic* constraint A_x of type AlgebraicExpression or a $differential$ constraint Dx of type DifferentialExpression. Each invariant and algebraic expression is a predicate over Q_V, each differential expression is a predicate over $Q_{V \cup d(V)}$. A flow f for a set of variables is a differentiable function from a closed interval of non-negative reals [0, δ] to Q_V. δ is the *duration* of the flow. The set of flows for V

is $\mathcal{F}_V$. A flow is permitted by the mode if for every t in the domain of f, every variable in $f(t)$ satisfies all constraints in $Cons$.

T is a finite set of transitions, i.e. the ModeTransitions in a Mode. A transition has the form (e, α, x), where $e \in E \cup SM.X$, $x \in X \cup SM.E$ and α a relation from Q_V to Q_V. α is the action of the transition, denoted by an RTExpression.

Every Mode has an *identity* transition from de to dx. There are no other transitions of which dx is the target. A transition that originates at dx of a Submode is called *group transition*, as it can be chosen independently of the current active Submode. If a submode has been exited by a group transition, it must be entered again through its default entry point to resume the interrupted execution. A Mode cannot be blocked at any of its non-default control points, i.e. for every e of M that is not de of M or dx of one of the Submodes of M, the union α_e of all actions of the transitions originating at e is complete; for every s there is t such that $(s,t) \in \alpha_e$.

A Mode is called $leaf$ Mode if $M.SM = \emptyset$. Leaf Modes perform continuous steps according to their constraints.

A Mode can engage in discrete or continuous behavior, by either taking some transition and changing variables accordingly to its action, or by residing in a Mode and letting time pass, updating the analog variables according to their flow constraints. We describe the state of a Mode M by a pair (c,s) where c denotes the control location (a control point of the Mode), and s is a valuation of the variables of M, i.e., $s \in Q_{M.V}$, also referred to as data state of M. While one can follow an execution of a Mode rather intuitively, a formal semantics require some technical extras to express precisely the richness of possibilities of behavior of hybrid systems. In particular to cover the possibility of interrupting an execution and resuming it later, a closure operation is applied to Modes, adding transitions to it which simulate (possibly a multitude of) internal steps of the system.

The closure $c(M)$ of a Mode $< E, X, V, SM, Cons, T >$ is defined as $< E, X, V \cup h, c(SM), Cons, c(T) >$, where $h \notin V$ is a new (local) variable used as *history* of the Mode. $c(SM)$ is the set of closed submodes of M and the closure $c(T)$ of the transition set T is obtained as follows:

- for every $x \in SM.dx$ a transition (x, id, dx) is added, i.e. from every default exit point of a Submode we have a transition to the Mode's default exit point which is always enabled and which does not change any variable.
- for every $e \in SM.de$ a transition (de, α_x, e) is added with $(s,s) \in \alpha_x$ iff $x \in N.E$ for some Submode $N \in SM$ and $s[h] = N$. This means that we introduce a new transition from a Mode's default entry point to the entry default point of each of its submodes, and these transitions do not change any variable. Yet, these transitions are only enabled if the value of the history variable h is the name of the Submode it leads to.
- every transition $(e, \alpha, x) \in T$ is replaced by (e, α', x), where $(s,t) \in \alpha'$ iff $(s[V], t[V]) \in \alpha$, and
  - if $x \in N.E$ for some Submode $N \in SM$, then $t[h] = N$, otherwise $t[h] = \epsilon$.
  - if $e \in N.X$ for some Submode $N \in SM$, then $s[h] = N$, otherwise $s[h] = \epsilon$.

  All transitions of M are augmented such that the history variable h always stores the information which Submode is currently accessed, i.e., which Submode is *active*.

For the closure of a Mode, one can describe its operational behavior now in terms of a continuous relation R^C and discrete relations $R^D_{c_1,c_2}$, defined for each pair $c_1 \in E$ and $c_2 \in X$.

To avoid ambiguity, continuous operation can only take place when the control state of the Mode is its default exit dx. Then the relation R^C gives for every data state s of that Mode a set of flows from this state, which complies to the constraints of that Mode as well as to the constraints of its active Submode. Formally, for Mode M, state s and flow f, $(s,f) \in R^C$ iff f is permitted by M and, if N is the active Submode, $(s[N.V], f[N.V]) \in N.R^C$. Furthermore, to guarantee consistency, we have that whenever $(s,f) \in R^C$, also $f(0) = s$ holds, i.e. the flow really starts with s. The set of flows $\mathcal{F}_s = \{f | (s,f) \in R^C\}$ is prefix-closed.

For each pair of control points $c_1 \in E \cup SM.X$ and $c_2 \in X \cup SM.E$ of a Mode, with c_1 an entry point of that Mode or an exit point of a Submode, and c_2 an exit point of the Mode itself or an entry Mode of a Submode, a relation $R^D_{c_1,c_2}$ is defined describing the path from c_1 to c_2 by the state changes induced along this path (no time passes taking a discrete transition). For a leaf Mode this is identical to the relation α belonging to the transitions allowed in that Mode, for a Mode that contains Submodes, such a path is composed out of transition steps of the Mode itself, and of steps which are taken within a Submode from an entry point of that Submode to an exit point of it.

The relation $R^D_{e,x}$ describes the state changes induced along a path from an entry point $e \in E$ of the Mode to an exit point $x \in X$ and is called a *macro-step*, as this may include transitions of the Mode itself (which are considered to be a *micro-step* in an execution leading from e to x) as well as steps of its Submodes, which are macro-steps w.r.t. that Submode, but micro-steps for the enclosing Mode. Formally, we observe a micro-execution of M leading from $e \in E$ to $x \in X$ as sequence of the form $(e_0, s_0), (e, s_1), \ldots, (e_n, s_n)$ where $e_0 = e$, $e_n = x$, and for even i we have $((e_i, s_i), (e_{i+1}, s_{i+1})) \in T$, while for odd i we have $(s_i, s_{i+1}) \in N.R^D_{e_i, e_{i+1}}$ for some Submode $N \in SM$. Such a sequence contains alternating steps by the transitions of the Mode itself, and steps through the Submodes as defined by their respective discrete relation. For such an execution, we have $(s_0, s_n) \in R^D_{e,x}$.

The semantics of a Mode M are defined by a transition system $\mathcal{R}_M$ over the states of the Mode, where we have transitions

- of the form $(c_1, s_1) \stackrel{o}{\to} (c_2, s_2)$ if $(s_1, s_2) \in R^D_{c_1,c_2}$ with $c_1 \in E$ and $c_2 \in X$,
- of the form $(dx, s_1) \stackrel{f}{\to} (dx, s_2)$ if $(s_1, s_2) \in R^C$, the flow f is permitted on the interval $[0, t]$, $f(0) = s_1$, and $f(t) = s_2$,
- of the form $(x, s_1) \stackrel{\varepsilon}{\to} (e, s_2)$ for $x \in X$, $e \in E$ and $s_1[V_l] = s_2[V_l]$.

The latter type of transition represent an environment step w.r.t. the Mode, to capture the possible state changes while the Mode is not active. Thus an environment step starts at an exit point of the Mode, leads to an entry point of it, and does not change the local variables (including the history variable h) of that Mode.

An *execution* of a Mode M is a path through $\mathcal{R}_M$ of the form

$$(e_0, s_0) \stackrel{\lambda_1}{\to} (e_1, s_1) \stackrel{\lambda_2}{\to} (e_2, s_2) \ldots \stackrel{\lambda_n}{\to} (e_n, s_n)$$

where λ_i ranges over $\mathcal{F}_V \cup \{o, \varepsilon\}$. A trace of a Mode M is a projection of an execution of that mode onto its global variables; it is described as a sequence $s_0[V_g] \stackrel{\lambda_1}{\to} s_1[V_g] \stackrel{\lambda_2}{\to} s_2[V_g] \ldots \stackrel{\lambda_n}{\to} s_n[V_g]$, where transition labels of the form $f \in \mathcal{F}_V$ are replaced with $f[V_g]$. These trace semantics provide the base for defining the behavior of the enclosing agent.

Notation

Modes are visualized the same way as UML 2.0 StateMachines (see Fig. 28). The identity transition from *de* to *dx* is not visualized explicitly as every Mode has it. Parameters are given behind the name of the Mode in parentheses. The invariant is marked *inv*, the flow conditions with *flow*. As both are constraints, they are given in brackets.

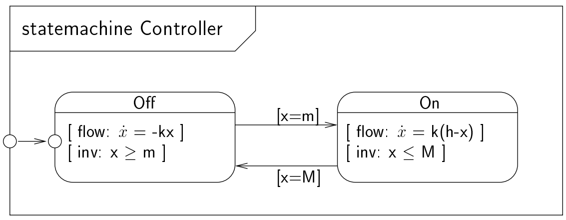

Figure 28: Notation for Mode

A.6.2 ModePseudostate

Description

ModePseudostate is an instance of Pseudostate. ModePseudostates are used as entry and exit points for Modes. Further, they connect multiple transitions into more complex ones.

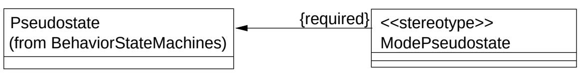

Figure 29: Stereotype for ModePseudostate

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- Each ModePseudostate is of kind ModePseudostateKind, i.e.
  self.kind implies oclIsTypeOf(ModePseudostateKind)
- A ModePseudostate is either entryPoint, exitPoint, defaultEntry, defaultExit, or junction.
  self.kind = (entryPoint or exitPoint or defaultEntry or defaultExit or junction)

Semantics

Semantics are given in Mode.

Notation

entryPoints are depicted as small circles on the border of a Mode (see Fig. 28) with an optional name attached to it. exitPoints are depicted as small solid black-filled circles on the border of a Mode with an optional name attached to it. defaultEntry points are not depicted explicitly as every Mode has exactly one; transitions to the defaultEntry point end at the boundary of the Mode. defaultExit points are not depicted explicitly as every Mode has exactly one; transitions starting at the defaultExit point start at the boundary of the Mode. junction points are depicted as small black-filled circles inside Regions.

A.6.3 ModePseudostateKind

Description

ModePseudostateKind is an instance of PseudostateKind. It is an enumeration of the following literal values:

- initial
- deepHistory
- shallowHistory
- join
- fork
- junction
- choice
- entryPoint
- exitPoint
- terminate
- defaultEntry
- defaultExit

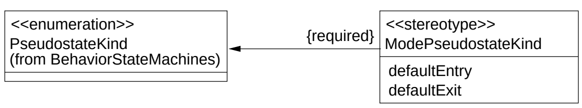

Figure 30: Stereotype for ModePseudostateKind

Attributes

No additional attributes.

Associations

No additional associations.

A.6.4 ModeRegion

Description

ModeRegion is an instance of Region. It contains ModeStates and ModeTransitions. Each Mode consists of one region. Orthogonal regions are not allowed.

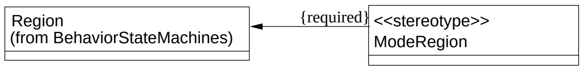

Figure 31: Stereotype for ModeRegion

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- All transitions are ModeTransitions, i.e.
  self.transition->forAll(oclIsTypeOf(ModeTransition))
- Regions are never extended, i.e.
  self.extendedRegion->size = 0
- Regions are never redefined, i.e.
  self.redefinitionContext->size = 0
- All vertices are either ModePseudostates or ModeStates, i.e.
  self.subvertex->forAll(v | v.oclIsTypeOf(ModePseudostate) or v.oclIsTypeOf(ModeState))
- All ModePseudostates in a ModeRegion are junction points, i.e.
  self.subvertex->select(v | v.oclIsTypeOf(ModePseudostate))->forAll(kind = junction)

Semantics

Semantics are given in Mode.

Notation

As orthogonal regions are now allowed, there is no notation for region. They are just the space inside a Mode.

A.6.5 ModeState

Description

ModeState is an instance of State. Each Mode consists of ModeStates. ModeStates are always Submodes.

Attributes

No additional attributes.

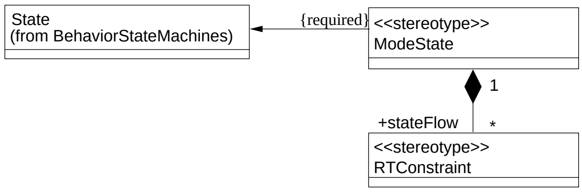

Figure 32: Stereotype for ModeState

Associations

- stateFlow: RTConstraint[*]

Constraints

- ModeState is never a composite state, i.e.
  self.isComposite = false
- ModeState is never orthogonal, i.e.
  self.isOrthogonal = false
- ModeState is never simple, i.e.
  self.isSimple = false
- ModeState is always a Submode, i.e. a submachine:
  self.isSubmachineState = true
- There are no connection point references, i.e.
  self.connection->size = 0
- There are no triggers as CHARON knows only global variables, i.e.
  self.deferrableTrigger:Trigger->size = 0
- There is no do activity, i.e.
  self.doActivity->size = 0
- There is no entry activity, i.e.
  self.entry->size = 0
- There is no exit activity, i.e.
  self.exit->size = 0
- ModeState will not be redefined, i.e.
  self.redefinedState->size = 0
- ModeStates have exactly one region, i.e.
  self.region->size = 1
- ModeStates are attached to a Mode, i.e.
  self.submachine.oclIsTypeOf(Mode) and self.submachine->size = 1
- There is at most one invariant of type RTConstraint, i.e.
  self.stateInvariant.oclIsTypeOf(RTConstraint) and
  self.stateInvariant.specification.oclIsTypeOf(InvariantExpression)
- ModeStates will not be redefined, i.e.
  redefinitionContext:Classifier->size = 0
- As each ModeState is a Submode, it is not a top-level Mode, i.e.
  self.submachine.isTopLevelState = false

Semantics

Semantics are given in Mode.

Notation

A ModeState is visualized in the same way as UML 2.0 States (see Fig. 28).

A.6.6 ModeTransition

Description

ModeTransition is an instance of Transition. It connects a target and a source ModePseudoState. ModeTransitions are taken due to guard constraints. A ModeTransition may have an associated effect that updates some variables.

A transition that originates at a default exit point of a Mode is called group transition. Group transitions are taken to interrupt the execution of that Mode. After that, the Mode must be entered again through the default entry point to resume the execution of the Mode.

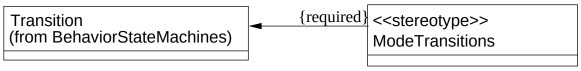

Figure 33: Stereotype for ModeTransition

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- There are no events in CHARON, therefore we do not have triggers, i.e.
  self.trigger->size = 0
- All guards are RTExpressions, i.e.
  self.guard->forAll(oclIsKindOf(RTExpression))
- All effects are ModeTransitionActivities, i.e.
  self.effect.oclIsTypeOf(ModeTransitionActivity)
- The source of a ModeTransition is a ModePseudostate, i.e.
  self.source.oclIsTypeOf(ModePseudostate)
- The target of a ModeTransition is a ModePseudostate, i.e.
  self.target.oclIsTypeOf(ModePseudostate)
- We do not replace transitions, i.e.
  self.replacedTransition->size = 0
- We do not refine transitions, i.e.
  self.redefinitionContext->size = 0

Semantics

Semantics are given in Mode.

Notation

ModeTransition is depicted by an arrow with open arrowhead (see Fig. 28). The guard constraint is given in brackets. The updateAction is separated from the guard by a slash.
The default transition from *de* to *dx* of each Mode is not visualized.

A.6.7 ModeTransitionActivity

In UML 2.0, Transitions can only have Activities as effect. We therefore need an Activity that updates variables as this is the only thing we do when performing transitions.

Description

ModeTransitionActivity is an instance of Activity (see Fig. 34). It updates variable values of Agents.

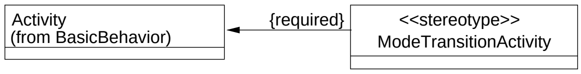

Figure 34: Stereotype for ModeTransitionActivity

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

- We use CHARON, therefore
  self.language='CHARON'
- The String given as *body* of the Activity must be an expression that updates variables V of an Agent: Let V_s be the set of variables in *body*, then $V_s \subseteq V$. Let Q_s be the set of valuations in *body*, then $Q_s \subseteq Q$ with Q valuations of Agent.

Semantics

Variables of the Agent that owns the Mode to that the ModeTransitionActivity belongs are updated.

Notation

ModeTransitionActivity has only a notation in context with ModeTransition. Therefore see ModeTransition.