Charon User Manual

(Version 0.9)

Rajeev Alur, Yerang Hur, Franjo Ivančić, Insup Lee, Oleg Sokolsky

Department of Computer and Information Science
University of Pennsylvania

January 28, 2002

Contents

Chapter 1

Introduction

A hybrid system is described in CHARON by a set of *agents* communicating over a set of shared variables in an asynchronous way. The agents may be grouped together in a hierarchical way into composite agents starting from the most primitive ones called atomic agents. Information flow inside a composite agent may be hidden to the outside world. The grouping of agents into composite agents gives the architecture of the hybrid system.

Atomic agents may be endowed with a set of parameters that can be instantiated in different ways. Thus an atomic or composite agent may also be understood as an architectural pattern that may be instantiated, i.e., reused in different contexts that match the pattern. For example, at a lower level, a robot may be understood as the composition of a sensing agent, a controller agent, and an actuator agent. At a higher level, one may consider a team of cooperating robots, communicating with each other in order to achieve a common goal.

The behavior of an atomic agent is given by a set of *modes* that are linked together by a set of transitions. Each mode represents a particular behavior of the agent and has an associated dynamics given by a set of algebraic and differential constraints. The dynamics may be further constrained by a set of invariants. Modes may also be grouped together in a hierarchical way to form composed modes starting from the most primitive ones called leaf modes. Moreover, each mode may declare its own set of local variables that is hidden outside the mode, but is accessible to its submodes.

In other words, a mode is a sequential, communicating, hierarchical state machine with well defined dynamics and interface to other modes. Variables of a hierarchical mode have scoping rules similar to scoping in structured programming languages. A mode may also have a number of parameters and be regarded as a behavioral pattern that can be instantiated.

Organization of the manual. We begin by presenting a simple but illustrative example of a hybrid system model in CHARON in Chapter 2. The example is a water tank in which the water level is controlled by a pump. The model is used to introduce the CHARON language and explain the modeling approach. We then use the same example to illustrate all the main components of the CHARON toolset in Chapter 3, starting with the visual specification editor in Section 3.1. Then we proceed to the main control panel and the textual editor in Section 3.2. Further, the visual simulator is described in Section 3.3. We conclude with the language reference guide that gives a detailed description of CHARON syntax in Section 4.1. The syntax described here corresponds to CHARON grammar version 0.6. Then, in Section 4.2, we describe the informal semantics of CHARON by means of a high-level description of a CHARON simulation algorithm.

More information concerning the application of the CHARON modeling approach to embedded systems can be found in [1]. The compositional semantics of CHARON has been laid out in [2].

Chapter 2

CHARON Models of Hybrid Systems

2.1 Agents and Architectural Hierarchy

A hybrid system is described in CHARON by a set of *agents*. Agents are autonomous entities that proceed independently of each other. Behavior of each agent is given in terms of a set of variables accessible by the agent. Variables of an agent can be either *global* or *private*. Global variables are accessible by other agents, while private variables are confined to the agent. A global variable is classified as either *read*, in which case the agent cannot update it, *write exclusive*, when only this agent can update it, and *write shared*, when more than one agent may update the variable. Read/write relationships between shared variables of different agents allow the agents to communicate asynchronously and define the information flow between agents.

Agents can be grouped together into composite agents in a hierarchical way. A composite agent consists of a number of subagents. When the subagents communicate using a private variable of the agent, the information flow is hidden from the agent's environment. Atomic agents do not have any internal structure and their behavior is given by modes (see Section 2.2 below). The grouping of agents into composite agents and the information flow between agents defines the architecture of the hybrid system.

Consider the model of a leaky tank, controlled by a pump. The pump has a sensor for the level of the liquid in the tank, but does not know the rate at which the liquid leaks out. The architecture of the model is given in Figure 2.1. The agent *LeakyTank* has two private variables; it is a closed system that does not depend on its environment. The subagent *Pump* represents the controller, which provides the variable *flow* and reads the variable *level*, provided by the tank. The subagent *LTank* has further structure and consists of the tank itself (subagent *Tank* and the subagent *Hole* that defines the leak in an opaque way. The variable *leak* is used to communicate between submodes. It is private in *LTank* so that the pump cannot know its value. Note that the agent *LTank* and its subagents use the variable *inflow* to denote input from the controller. Variable renaming is used to rename *inflow* to *flow* so that the agents can communicate properly.

The CHARON code corresponding the the model in Figure 2.1 is shown in Figure 2.2.

Note that the model fragment contains a definition of the agent *LTank*, which is instantiated as a submode within *LeakyTank*. The same agent can be defined once and instantiated many times. Agent definitions may be parameterized as seen in the definition of agent APump. When a parameterized agent is instantiated, the values for the parameters are provided in a way similar to

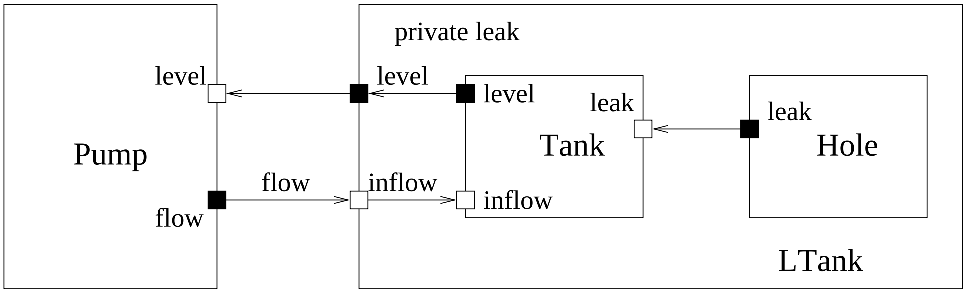

Figure 2.1: The data flow architecture for the Leaky Tank model

```
// ------------------------------------------------------------------------
// a leaky tank controlled by a pump

agent LeakyTank() {
  private analog real level, flow;

  agent pump = Pump( 5, 10 )
  agent tank = LTank( ) [inflow := flow]
}

// ------------------------------------------------------------------------
// the tank agent with a hidden leak

agent LTank() {
  read analog real inflow;
  write exclusive analog real level;
  private discrete real leak;

  agent tank = Tank()
  agent hole = Hole()
}

// ------------------------------------------------------------------------
// the pump agent

agent APump( int low, int high ) {
  write analog real flow;
  read  analog real level;

  init { flow = 0; }

  mode top = PumpMode( low, high );
}
```

Figure 2.2: Textual Charon model for Figure 2.1

the passing of call-by-value parameters in conventional programming languages.

2.2 Modes and Behavioral Hierarchy

Behavior of an atomic agent is given by a mode. Modes are hybrid state machines and can also be hierarchically organized, creating a behavioural hierarchy. An atomic mode contains a set of constraints that apply to the variables of the mode. There are three kinds of constraints: differential, algebraic, and invariant constraints (or simply invariants). An example of an atomic mode is *TankMode* that gives the behavior for the agent *Tank*, shown below. It has a single differential constraint that defines the flow for the variable *level* in terms of input variables inflow and leak.

```
mode TankMode() {
  read discrete real leak;
  read analog real inflow;
  write exclusive analog real level;

  diff { d(level) == inflow-leak }
}
```

A composite mode consists of a number of submodes. During execution, exactly one of the submodes is active. As the execution evolves, instantaneous mode switches can occur by taking discrete transitions between submodes. Mode switches give rise to the notion of *discrete thread of control* within a mode. Discrete control can enter and leave a mode via named entry and exit control points. Together with the global variables of a mode, its control points define the interface of the mode. Each mode has a default entry point *de* and a default exit point *dx*, which will be discussed in more detail below. In the visual representaion of modes, we will omit the names of the default control points to avoid clutter.

Figure 2.3 shows a composite mode that gives the behavior of the agent *Pump*. The mode consists of two submodes, *Maintain* and *Compute*. The mode *Maintain* describes the pump that supplies the flow of liquid into the tank. It has four submodes. In two of the modes, the flow is constant. When the pump is off (mode *off*), the flow is zero, and when the pump is on (mode *on*), the flow is given by the discrete variable *rate*. The other two modes correspond to the transient state of the pump being turned off or adjusted to the new rate. In Figure 2.4 we show a fragment of the textual version of the mode *Maintain* to illustrate mode instatiation. Submodes *on* and *off* are instances of the same mode *SteadyMode* with different parameter settings. Note also the use of external function `abs` in the guards of the transitions. Any static method of a Java class, standard or user-defined, can be used as an external function. User-defined classes are assumed to reside in the package `CHARON.user`.

Switches between submodes are performed according to guards on mode transitions. A transition can be taken when it is *enabled*, that is, its guard is satisfied. The execution of an enabled transition can be forced by the *invariant* of the mode. An invariant is a predicate over the mode variables. For example, the invariant of the mode *Maintain* is $clock \leq 1$. When the transition occurs, the action associated with the transition is executed. For example, the transition from mode *Maintain* to *Compute* (see Figure 2.3) is possible when the variable *clock* exceeds 1 and when

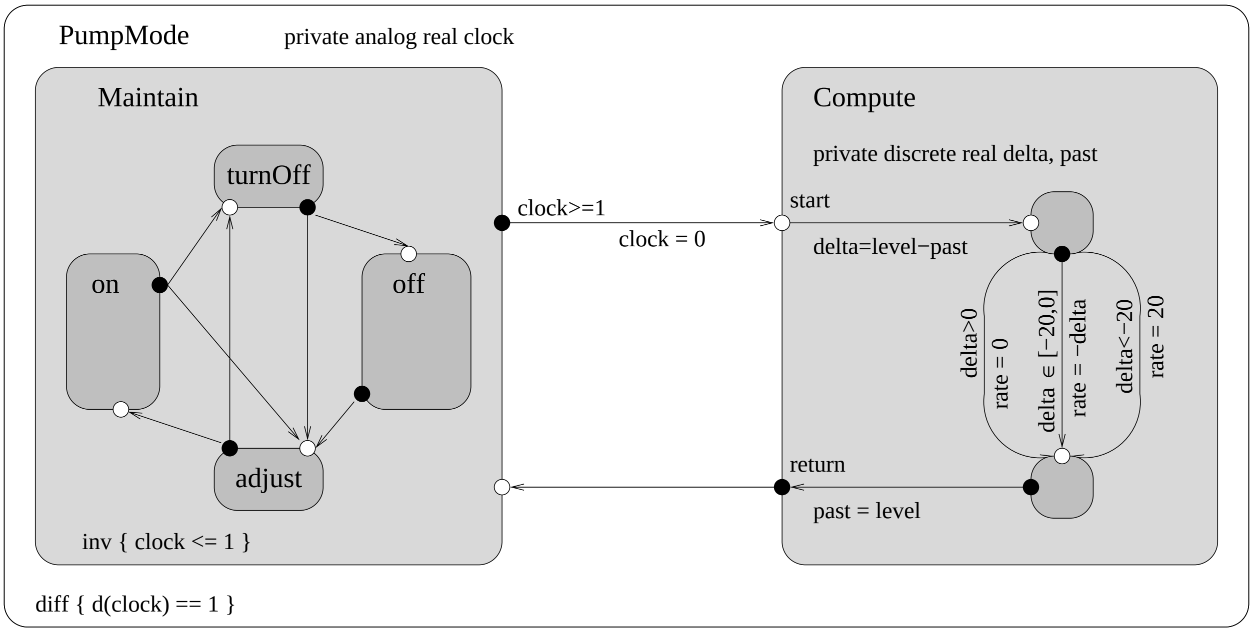

Figure 2.3: Hierarchical structure of mode *PumpMode* (transition guards in mode *Maintain* are omitted to avoid clutter)

```
mode Maintain( real tolerance, int low, int high ) {
  write analog real flow;
  read    analog real level;
  write   discrete int  hist;
  read    discrete real rate;

  mode on =      SteadyMode( tolerance, 0, high );
  mode off =     SteadyMode( 15, low, 10000000 );
  mode adjust =  TransientMode( tolerance, high );
  mode turnOff = TurnOffMode( low );

  trans from default to on      when true do {}
  trans from on      to turnOff when level > high  do {}
  trans from turnOff to off     when flow <= 0    do {}
  trans from off     to adjust  when level < low   do { }
  trans from adjust  to on      when java.lang.Math.abs(rate-flow) <= tolerance do { }
  trans from on      to adjust  when java.lang.Math.abs(rate-flow) > tolerance do { }
  trans from turnOff to adjust  when level <= low do { }
  trans from adjust  to turnOff when level > high do { }
}
```

Figure 2.4: Textual representation of mode *Maintain*

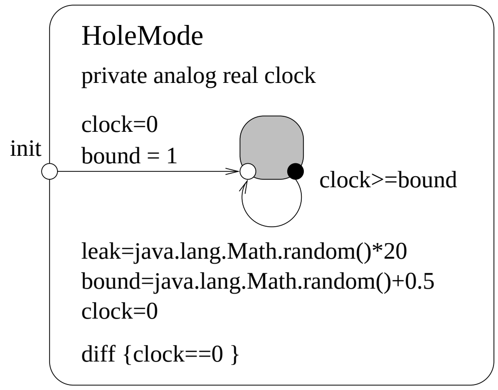

Figure 2.5: External objects in CHARON models

the transition is taken, *clock* is reset. Together with the invariant of *Maintain*, the guard ensures that the transition is taken immediately.

Since the leak rate is unknown to the pump controller, it needs to compute an estimate of the flow necessary to maintain the level. Therefore, periodically, the controller calles the estimation routine, implemented by the mode *Compute*. It is a purely computational mode, whose executions are assumed to be instantaneous. Much like a procedure in a high-level programming language, *Compute* has an entry point *start*, through which the computation is called. The structure of the mode is, essentially, the flow of computation of the procedure. When the computation is completed, control is returned to *Maintain* via the exit point *return*.

The example also illustrates the notions of *preemption* and *history retention* that are important in the CHARON modeling approach. When it is time to perform recomputation, the discrete control leaves the mode *Maintain* via the default exit point. This preempts the execution in any of the submodes and the location of discrete control is remembered. Then, when the discrete control is returned via the default entry point, the execution will resume from the point where it was preempted. For example, if the pump was in the *turnOff* mode when the recomputation commenced, after the recomputation is complete, the pump will still be in the *turnOff* mode.

By contrast, when a mode voluntarily releases control, as in the mode *Compute* after recomputation is complete, no history is kept, and the execution will start next time from the beginning when the *start* control point is invoked.

Finally, we show the implementation of the non-deterministic unobservable leak in the tank. The mode *HoleMode* (Figure 2.5) chooses randomly the new value for the leak rate, and also the next time the leak rate will be updated. This mode illustrates the ability of CHARON models to use uncertain input in a model by means of external components. In this case, a random generator provided in the Java math library is used twice in the action block of the transition of the mode *HoleMode*.

Chapter 3

Charon Toolkit

3.1 CHARON Visual Specifications

Since we have presented the example largely in terms of the pictorial representation of Charon models, we begin the presentation of the Charon toolkit with the visual editor. It allows the user to draw agent and mode definitions at a given level of hierarchy. The interpreter of the visual input translates the specification into text-based Charon source code using an intermediate XML-based representation.

The tools consists of overlayed visual editor panels that allow to manipulate agent and mode structures. In addition, on the left-hand side there are the tools panel, the navigation panel, and the definitions panel. The navigation panel shows the structure of the current project. A component can be expanded to show its subcomponents, and double-clicking on the component opens it in the visual editor window on the right.

The visual input tool is depicted in Figure 3.1, showing one level from the behavioral hierarchy of the leaky tank example. Agents are represented as boxes with interface variables represented as squares. Arrows connecting interface points show the data flow between subagents. The user can explore the lower levels of hierarchy by clicking on the sub-agents or via the navigation panel. The definitions panel allows to view and manipulate parameters, variables, and constraints or the current agent or mode.

In the editor window, subagents and sumodes are added by dragging a definition icon from the navigation panel. The interface of the submode or subagent is drawn by the tool automatically according to the definition. Additional editing operations are enabled via the tools panel. For the agent view, the user can add arrows between the interface points.

Modes are shown as rounded boxes, with circles representing control points. Arrows between control points indicate control flow between modes. By convention, default control points are not shown, and a transition to or from a default control point is attached directly to the box. The guard and the action block of a transition can be seen by clicking on the transition. Tools of the mode view offer the possibility to add an entry or an exit point, and to draw a transition between two control points.

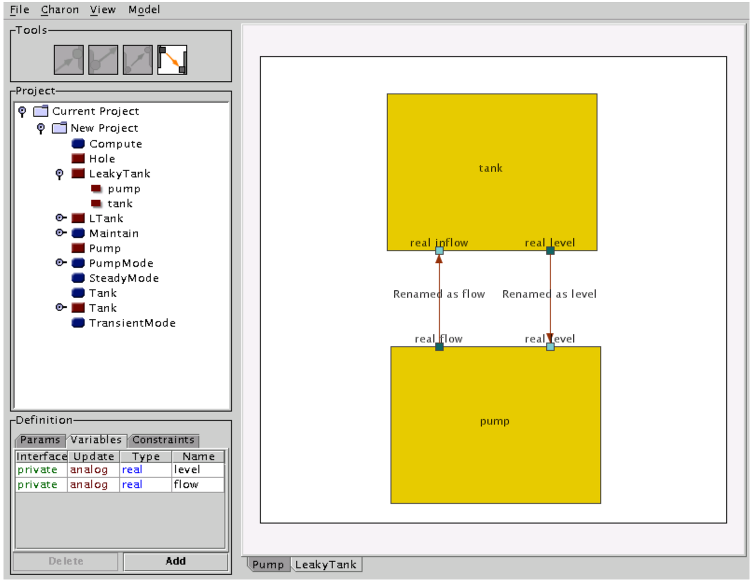

Figure 3.1: The visual editor of CHARON

3.2 CHARON Control Panel and Textual Editor

Overview

The centerpoint of the CHARON toolkit is the control panel. The panel allows the user to create, navigate, and manipulate large models. The panel incorporates an editor, a parser, and a type checker for textual CHARON models. It also provides connections to other components of the CHARON toolkit, such as the visual editor, simulator, etc.

The control panel, shown in Figure 3.3, consists of a number of menus and toolbar, and two main panels. The left panel displays the structure of the current model in a tree format. The user can expand the nodes to see their contents. The panel is overlayed with the list of files that comprise the model. The right-hand side panel consists of editor windows where changes to the model can be made.

Project management

CHARON control panel organizes models into projects. Each project is associated with a single model, which may be split into several files. When a new project is created using the `New Project` item from the `File` menu, the user specifies the project name and the location for the project. A new directory in the specified location is created with the given project name. The project directory has a number of subdirectories, for example, subdirectory `traces` stores the simulation results. In addition to the predefined directory structure of the project directory, an empty project file is created, which has the project name and extension `.prj`. The project file is the list of CHARON files that are included in the project. The `Project` menu lets the user add new files to the project, remove files, and rebuild the model following the changes.

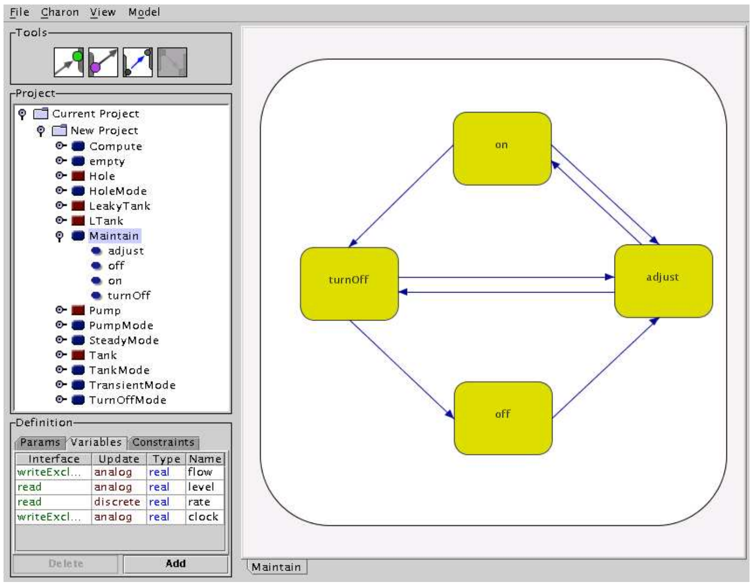

Figure 3.2: Modes view of the visual editor

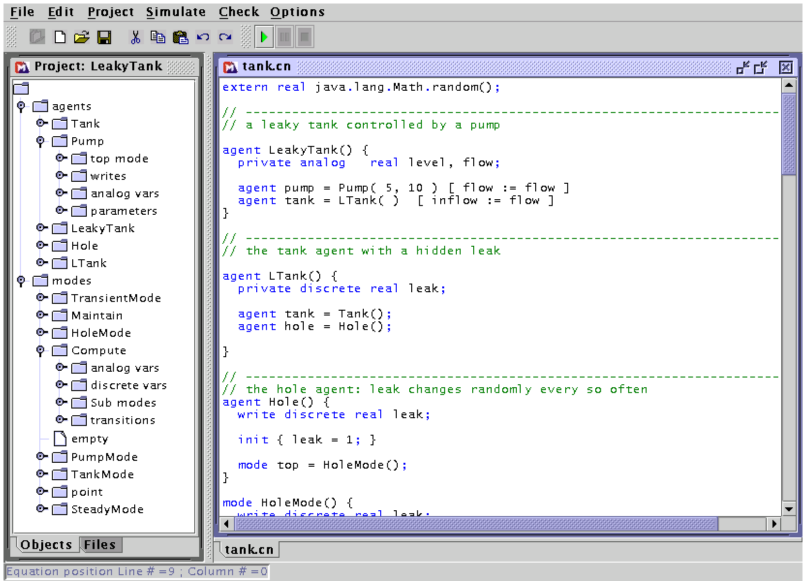

Figure 3.3: CHARON control panel

Textual editor

The textual editor allows the user to edit CHARON models. It is a simple but fully functional editor that allows the standard editing operations such as copy, cut, and paste, as well as undo and redo. The editor is equipped with a light-weight CHARON parser, which provides syntax-driven aid to the user. The keywords in the editor window are recognized and highlighted in blue and syntactically incorrect constructs are highlighted in red. In addition, the parser and type checker produce error lists that are linked to the model text so that selecting an error message highlights the corresponding section in the editor window.

Analysis of CHARON models

The CHARON toolkit currently allows two ways of analyzing properties of the model. The first way is checking of invariant assertions during the simulation, which will be discussed in Section 3.3. The second analysis tool is reachability analysis based on the predicate abstraction. The predicate abstraction tool is distributed separately from the main toolset and is currently available only on Solaris and Linux platforms. The reachability analysis method is applicable only to linear hybrid systems. Automatic translation of linear CHARON models into the input language of the reachability analysis tool is available via the `Reachability` submenu of the `Check` menu of the control panel.

3.3 CHARON Visual Simulator

The toolkit includes the simulator for CHARON models. The simulator can be used in two modes: interactive visual simulation and background simulation. Background simulation produces a trace of the execution that can be visualized and analyzed later. During an interactive simulation run, a plotter is executed concurrently with the simulator and displays the simulation trace as it evolves. The plotter can also display a stored simulation trace.

All traces are stored in the subdirectory `traces` of the project directory. Each simulation trace is stored in two files with the following naming convention. If `A` is the name of the project, then the file `AgSim.rep` keeps the values of the variables in the model during the simulation. Discrete transitions that occurred during the execution are stored in the file called `trLogAgSim.rep`. For reference, the tool keeps a history of simulations. Up to ten versions a simulation trace are stored, identified by their numbers. The unnumbered files (`AgSim.rep` and `trLogAgSim.rep`) are always the latest version. The version with the least number (`AgSim1.rep` and `trLogAgSim1.rep`) is the oldest, and version numbers increase for successive newer versions.

The simulator also produces the file called `graphAgSim.rep` that stores the trace in the format accepted by the plotting program `xgraph`. This feature is superseded by the plotter of the toolkit.

Running the simulation. Before a CHARON model can be simulated, a simulator has to be generated. The generation process produces a set of Java classes for modes and agents of the model. The generated classes are executed together with the simulator core classes and external Java classes referred in the model to produce the complete simulator. Until the simulator is generated, all other simulation controls on the CHARON control panel are disabled. The toolkit keeps track of the changes to the model and disables the simulator once the project has been modified.

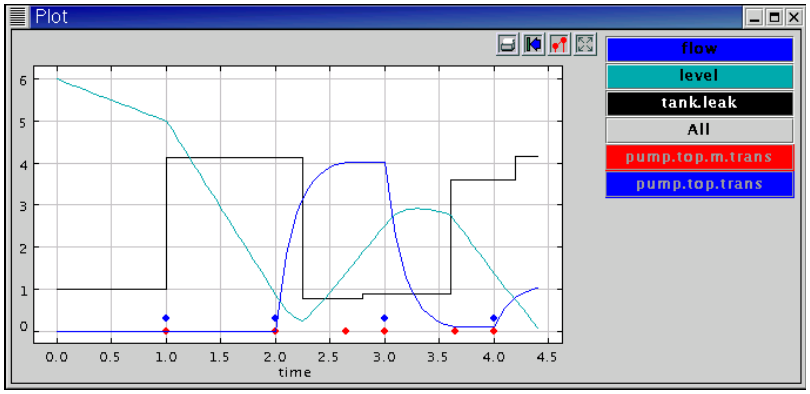

Figure 3.4: CHARON plotter

The background simulation is started via a menu item in the `Simulate` menu of the control panel. Once started, a background simulation cannot be interrupted. Because of the fixed naming convention, two concurrent simulations of the same project, whether background or interactive, cannot proceed simultaneously.

For controlling interactive simulations, the CHARON control panel includes the simulation toolbar with buttons to start, pause, and stop a simulation. When there is a generated simulator, the start button is enabled and the other two buttons are disabled. Once a simulation begins, the start button is disabled and the pause and stop buttons are enabled. If the simulation is interrupted, the simulator stops before the next simulation step. However, since the plotter proceeds asynchronously from the simulator, it will continue executing until the buffered simulation results are displayed.

Assertion checking. The simulator checks invariant assertions after every simulation step. Assertions can be embedded into any mode or agent definition the CHARON model. An assertion is a predicate over the variables of the mode or agent and is supposed to be invariantly true in any execution. For example, in the leaky tank example, the following assertion in the `TankMode` states that the level can neither exceed the capacity of the tank nor become negative:

```
assert { 0 <= level and level <= 15 }
```

During simulation, the simulator will check assertions in all agents and active modes and terminate the simulation if a violation is detected. The error message will identify the instance of the mode or agent where a violation has occured and the time of the violation:

```
Property violation is detected at time 4.449999999999996.
[agent] tank
[mode] tank.tank.top.
[property] name: asstTankMode0, id: 1
```

Plotter. The plotter panel is shown in Figure 3.4. The plotter is built around the `Plot` class from the Ptolemy project [3]. The functionality of the plotter is extended with the capability to change the information displayed by the plotter. When many variables are plotted during the simulation, all are displayed at the same scale. As a result, variables with the small range of values may be

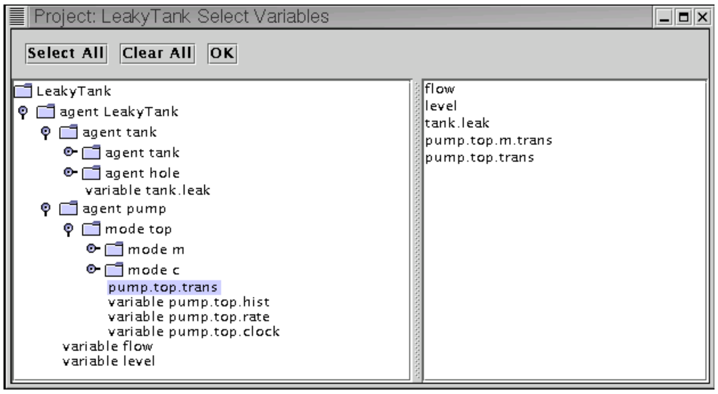

Figure 3.5: Variable selection dialog

difficult to see in the plotter graph. Furthermore, variables with similar dynamics may obscure each other in the graph. The Charon plotter allows to view all the variables selected to be displayed, or concentrate on one chosen variable. The right hand side of the plotter panel contains a color-coded list of the names of variables in the plot and one additional button labelled `All`. Selecting a variable from the list shows the graph of that variable drawn to appropriate scale. Selecting the `All` button shows the graph of all variables selected for display. The plotter can also display transitions of selected modes. Transitions are shown as dots at the bottom of the graph. Currently, transitions can be displayed only when a stored trace is viewed, but not during interactive simulation.

Selecting variables for display. When a large model is simulated, displaying all the variables that take part in the simulation may be inefficient and counterproductive. The Charon simulator lets the user choose the variables that are to be displayed. The selection dialog box can be opened through the `Simulate` menu of the control panel. The window is shown in Figure 3.5. The left panel is a tree view of the model. When a new level in the hierarchy is opened, the variables defined in that agent or mode are showed. The right panel is a list of selected variables. A variable is selected by double-clicking on the variable name in the left panel. A selected variable can be removed from the list by double-clicking on its name in the left panel again. The user can select all variables to be displayed by using the `Select All` button in the selection dialog or simply by skipping the variable selection step.

In addition to variables, discrete transitions of the desired modes can be selected. In each mode where there are transitions defined, a special entry `trans` is shown in addition to the variables of the mode. Selecting this entry will make the plotter display all occurrences of transitions within this mode.

Limitations of the simulator. Currently, not all language constructs are supported by the simulator. In particular, the simulator assumes that a mode defines only a single flow from any state. Because of this, inequalities cannot be used in differential and algebraic constraints.

History retention is currently not supported. If the execution needs to return to the point where it was interrupted, a history variable can be added to the manually and manipulated explicitly by the transitions. See the implementation of the mode *Maintain* of the leaky tank example in the Appendix.

Channels, allowed by the Charon syntax (see the Charon language in Section 4.1), are not implemented by the simulator. The only supported mode of communication is via shared variables.

Only one agent is allowed to update a variable during a continuous flow, however any number of agents can update a variable in a discrete fashion.

There is currently no GUI support for resolution of non-determinism, and the user has to resolve non-determinism manually. If the simulator encounters multiple enabled transitions, the simulation is interrupted and a message is printed out, enumerating the available transitions and asking the user to select one of the transitions. The transitions are identified by their destinations. The user enters the number of the desired transition.

Chapter 4

CHARON Language Reference

4.1 The CHARON language

This section presents the detailed syntax of the CHARON language, along with the consistency requirements for CHARON models. The consistency requirements ensure that the semantics of a model is well-defined, that is, it possesses a computable set of behaviors.

4.1.1 General notes on syntax

Identifiers. CHARON supports the standard convention on names. Identifiers are strings of arbitrary length, containing letters, numbers, and the underscore character, and starting with a letter. The names are case sensitive.

Keywords. The list of CHARON keywords is given below. Keywords cannot be used as identifiers. Keywords are case insensitive. Note that some of the keywords in the list below are not currently used in CHARON syntax, but are reserved for future extensions of the language.

`agent`, `alge`, `analog`, `and`, `assert`
`bool`,
`channel`, `cold`,
`default`, `diff`, `discrete`, `do`,
`entry`, `exclusive`, `exit`, `extern`,
`false`, `from`,
`hide`, `hot`,
`in`, `init`, `int`, `inv`, `isempty`,
`macro`, `mode`,
`of`, `or`,
`private`,
`read`, `readwrite`[1], `real`, `receive`,
`send`, `struct`,
`to`, `trans`, `true`,
`when`, `while`.

[1]Keyword `readwrite` is deprecated. It is kept for backward compatibility with early versions of CHARON language.

In all examples below, different fonts are used to represent `keywords` and *identifiers*.

Comments. CHARON supports both styles of Java/C++ comments: `//` ... and `/*`... `*/`.

Model structure. A CHARON model consists of non-empty sets of mode and agent definitions and a possibly empty set of declarations of external objects. They are described in Sections 4.1.4, 4.1.5 and 4.1.6, respectively. Modes and agents interact through shared variables and channels, which we describe in the next section.

Definitions and instances. We distinguish between definitions of modes and agents and their instances. The relationship between them is similar to classes and class instances (objects) in object-oriented programming. A definition of a mode or agent describes its structure and behavior, possibly in terms of a set of parameters. All definitions are global and no nesting of definitions is allowed.

A definition of, say, a mode can be used by another mode or agent to create a subcomponent, which will be an instance of the mode. To create an instance, a set of values are given to the parameters in the mode definition, and the instance is assigned a name by which it can be referred to by the instantiating mode or agent. To continue the analogy with object-oriented programming, instantiation is similar to the use of `new` in Java or C++.

Semicolons. Semicolons in CHARON are optional and may be used to conclude variable declarations and statements, such as assignment statements and instantiations of modes and agents. When in doubt, it is always safe to omit a semicolon. However, the development team thinks they enhance readability of CHARON models and also let C and Java programmers feel more comfortable.

4.1.2 Variables and channels.

Variables. Every variable in CHARON is assigned a type. Types are a subset of Java types and are either primitive or user-defined. Primitive types are integer (`int`), real (`real`), and boolean (`bool`). User-defined types can be either defined within the CHARON model or externally. External user-defined types are formed according the the Java typing convention. They are useful when referring to external objects.

In addition, variables are classified as either discrete or analog. Intuitively, analog variables are updated continuously (such as time, temperature, position of a robot, etc.). By contrast, discrete variables represent the state of a digital controller. They are updated by instantaneous (discrete) events and retain their values between events. Note that instantaneous events can modify the analog variables as well as discrete ones. However, discrete variables are modified *only* in a discrete fashion. Keywords `discrete` and `analog` are used to distinguish between the two kinds of variables and are referred to as *update specifiers*. An update specifier may be omitted from a variable declaration. The default value is `analog`.

Access restrictions. Variables are also classified according to access restrictions. A mode or agent may be able to read and write a variable, but may also be restricted to either reading or writing. Also, other agents or modes may be able to access the variable, or it may be internal to the agent or mode (but accessible to its subagents or submodes). *Access specifiers* are `read`, `write`, and `private`. An access specifier cannot be omitted from a variable or channel declaration. The `read` specifier means that the variable cannot change the value by the agent (see 4.1.4 for details). The `write` specifier means that the variable can be both read and updated by the agent. Write variables can further be specialized as `shared` or `exclusive`. A write shared variable can be updated by multiple agents, while write exclusive variables can be updated by only one agent. The specialization can be omitted, and a write variable is write shared by default. Finally, the `private` specifier limits the scope of the variable to the mode or agent that contains the declaration, and its subcomponents. Note that a variable is accessible to all agents within its scope. In this sense all variables are "read shared." Communication via shared variables is the primary means of communication between CHARON agents.

Data structures. Complex data types in CHARON are collections of named fields. Each field has a primitive or a complex type. Recursive data structures are currently not supported. When a variable of a complex type is declared, all of its fields share the same attributes and access restrictions. Consider the following example:

```
struct realpair {
    real f1;
    real f2;
}
```

Then, if a variable declared as `read analog` realpair p, both *p.f1* and *p.f2* will be treated as read analog variables.

Channels. In addition to shared variables, agents can communicate through typed channels of limited capacity. A channels is a queue of messages, so that one agent can write into the end of the queue and another can read from the beginning of the queue. Unlike variables, channels provide point-to-point communication. That is, only one agent can read a value from the channel; once it is read, the value disappears from the queue. Channel declarations have the same access specifiers as variables, but no update specifiers.

Channel overflow policies. Each channel has an associated overflow policy. When an agent writes into a full channel, the agent is not blocked. Instead, depending on the overflow policy for the channel, which can be either "cold" or "hot", one of the following happens. With the "cold" the new value is dropped and the data in the channel queue stays unchanged. Or, in the case of "hot" policy, the first (the oldest) value in the queue is dropped and the new value is inserted at the end of the queue.

Variable declarations. The syntax for variable and channel declarations is the same for both modes and agents:

$\langle$access specifier$\rangle$ $\langle$update specifier$\rangle$ type $\langle$list of names$\rangle$

or

⟨*access specifier*⟩ `channel`[size] `of` type ⟨overflow specifier⟩

For example, all of the following are valid declarations:

`read discrete int` *i1, i2*;
`write discrete real` x;
`write analog` robot.Position p;
`write channel`[25] `of real` *ch_out*
`private int` *distance*

On the other hand, the following two declarations are disallowed since only real-valued variables may be continuously updated. Note that variables of user-defined types, such as robot.Position, are allowed to be declared `analog`, because they can be collections of real variables.

`read analog int` n;
`write analog` bool *flag*;

4.1.3 Expressions

Expressions in CHARON models can be of an arithmetic or boolean type. Expressions are formed by applying arithmetic or boolean operators to variables and constants. Arithmetic operators include $+$, $-$, $*$, $/$, and % with the expected meanings. Boolean operators include relational operators and logical conjunction, disjunction, and negation. Relational operators are `==`, `!=`, `<`, `<=`, `>`, `>=`. Conjunction can be written either as `and` or `&&`. Disjunction is written as `or` or `||`. Negation is either `~` or `!`.

In addition to relational constructs, primitive boolean expressions are the two boolean constants `true` and `false`, and the predicate `isempty`, which is used to tell the status of a channel. Expression `isempty`(c) returns `true` if the channel c does not contain any messages and `false` otherwise.

4.1.4 Modes

A *mode* describes behavior of a system component with a single thread of control. Behavior is given by a hierarchical hybrid state machine. A mode contains a set of submodes connected by transitions. Such state machine describes the position of the discrete control in the component. Discrete control enters the mode through one of its entry point and exits through one of the exit points. A mode that has control is called active. While inside a mode, control is passed along the transitions of the mode from one submode to another. Transitions have *guards* and *actions* associated with them. Before a transition can happen, its guard must be true. When a transition happens, the variables of the mode are updated by the action of the transition. While the control resides in a submode, the variables are updated continuously with the passage of time in compliance with the constraints given by the submode and the enclosing mode.

Structure of a mode. A simple example of a mode, showing all kinds of constructs allowed in modes, is given below. The mode header, introduced by the keyword `mode`, gives a name and a list of typed parameters for the mode. The mode body consists of a control point declarations, variable declarations, and non-empty sequence of statements, which include mode instantiations, differential and algebraic constraints, invariants and transitions. We will explain each of these in turn, using the example below.

```
mode M( int p1, real p2 ) {
    // control point declarations
    entry in;
    exit out;
    // variable declarations
    private analog real clock;
    write analog real skew;
    // mode statements
    mode M1 = N(p1);
    diff { d(clock) == 1; }
    alge { skew == 0.9 * clock; }
    inv { clock < p1 }
    trans setClock from in to M1.in
        when true do { clock := p2; }
}
```

Parameters. Parameters of a mode are given as a comma-separated list of names with their types. Parameters can be used in the statements of the mode in the same way as `read` mode variables. Mode parameters are constants that are applied when the mode is instantiated before it can be used. The closest notion from the domain programming languages are the pass-by-value parameters to the constructor of a class. In the example, we have two parameters, *p1* of type `int` and *p2* of type `real`. When a mode is instantiated, concrete values for its parameters are specified. Since all modes are instantiated statically before execution begins, parameters must be evaluated during initialization.

Control points. Control points of a mode serve as the interface between the mode and its environment. Control points are classified into `entry` and `exit` points. They are analogous to entry and exit points of procedures in programming languages. An execution of a mode begins when it gains control at one of its entry points and ends when the control is returned to its environment at one of its exit points. In the example we have one entry mode *in* and one exit mode *out*. In addition, every mode has a default entry point and a default exit point. The default control points do not have names, but can be referred to using the keyword `default`. Control points are used in mode transitions as described below.

Variable declarations. Syntax for variable declarations in both modes and agents is described in Section 4.1.2. In the above example, there are two variables in mode *M*, both of type real. One of these variables is private, that is, not visible from the outside. The other variable can be updated by the mode *M* and can be read by modes and agents in the environment of *M*.

Mode instantiation. A mode can have a number of submodes. Each submode is a mode instance, created by an instantiation statement. The instantiation in the example:

```
mode M1 = N(p1);
```

creates a mode instance *M1*, assuming there exists a definition of mode *N* with a single parameter that matches the type of *p1*. The type conversion rules allow the parameter to be of type `int` or `real`.

Differential and algebraic constraints. Constraints define admissible flows for analog variables. That is, they describe how analog variables can change with time while the mode is active. All constraints take the form:

$\langle$controlled entity$\rangle$ $\langle$relation$\rangle$ $\langle$expression$\rangle$

A controlled entity is either a variable of the mode or its first time derivative. If v is a variable, then $\mathrm{d}(v)$ denotes its derivative. A constraint is called *algebraic* if its left-hand side is a variable and *differential* if it is a derivative. They are syntactically distinguished by keywords `alge` and `diff`, respectively. In our example, there are both differential and algebraic constraints.

`diff` $\{\ \mathrm{d}(clock) == 1;\ \}$
`alge` $\{\ skew == 0.9 * clock;\ \}$

The first constraint says that the variable *clock* indeed behaves like a clock, that is, changes at the same rate as time. The other constraint makes variable *skew* a clock with constant skew: its value is always a constant fraction of the value of the true clock.

While both constraints in this example are equalities, it need not be true in general. Inequalities can be used, and there can be more than one constraint for each controlled entity. We assume, although the current implementation does not attempt to check it, that the set of constraints is consistent; that is, it allows a non-empty set of solutions.

Invariants. Invariants specify when control can reside in the mode. Each invariant is a boolean expression over the variables of the mode. When the expression returns `false`, control has to leave the mode through one of its enabled transitions (if no transitions are enabled, the CHARON system is deadlocked). In the example, M has the invariant `inv` $\{\ clock < p1\ \}$. An invariant can be omitted in a mode, in which case it defaults to `true`; that is, the mode can be active indefinitely.

Transitions

A mode can have transitions between its submodes and/or its environment. Transitions represent discrete events in the mode such as transfer of control and can modify mode variables. Transitions connect control points of the mode and its submodes. Each transition has a guard associated with it; guards are represented by boolean expressions. A transition can be executed only if it is *enabled*, that is, its guard evaluates to `true`. In addition, a transition has an update block, which consists of assignments to mode variables and read/write operations on channels of the mode.

Update blocks. An update block of a transition, in general, contains a sequence of assignment statements and send/receive statements. A block can update a variable at most once (a variable is updated if it is on the left-hand side of an assignment or in a receive statement). Channels may be read or written multiple times in a block.

The syntax for send and receive operations is as follows:

```
send(⟨channel⟩, ⟨expression⟩)
receive(⟨channel⟩, ⟨variable⟩)
```

In its simple form, a transition connects two control points. There are four classes of transitions:

- *entry* transitions that lead from an entry point of a mode to an entry point of one of its submodes. Entry transitions are used to initialize private variables of the mode.
- *exit* transitions that lead from an exit point of a submode to an exit point of the mode.
- *internal* transitions that lead from an exit point of one submode to an entry point of another (possibly, the same) submode.
- *pass-through* transitions lead from an entry point to an exit point of the mode.

A transition may be given an optional name, which can briefly describe its purpose. In the example, there is an entry transition *setClock*:

```
trans setClock from in to M1.in
    when true do { clock := p2; }
```

This transition goes from an entry point *in* of M to an entry point *in* of $M1$. It is always enabled, that is, has guard `true` and assigns a value to *clock*, which is an analog variable that is updated here in a discrete fashion.

A more general form of a transition may involve multiple destination points, each with its own guard. In addition, there may be a *group guard*, which must be satisfied before any of the branches are considered. An example of such transition is given below:

```
trans from M0.out when isempty(c)
    to M1.in when a==1 do { send(c,1); }
    to M2.in when a==2 do { send(c,2); }
    ...
```

This transition may go to the submode $M1$ or $M2$, depending on the value of a, but the whole transition is possible only when the group guard is satisfied, that is, channel c does not have any messages. If the group guard is satisfied, then at least one of the other guards in the transition must be satisfied as well.

Control points. Control points in transitions are identified as pairs of names, separated by a dot. The first name is the name of a mode, the second name is the name of a control point in that mode. In the example above, entry point *in* can also be addressed as *M.in*. If a mode name is omitted, the current mode is assumed. If a control point name is omitted, the default entry or exit point is assumed. At least one of the names should be specified. The default control point is identified by the keyword `default`. M.`default` is the default control point of M. Since the default control point is used often, shorthand notations are allowed: M and `default` (used within the body of M) both refer to the default control point of M.

Note that the two default control points of a mode are identified by the same keyword, but there is no confusion possible. In a `from` clause, a default control point is the entry point of the mode itself or an exit point of a submode. In a `to` clause, the roles are reversed.

Consistency requirements for a mode

We assume that all specifications are type correct. That is, every expression appearing in invariants, constraints, as well as guards and update blocks of every transition, can be assigned a type based on the types of variables and operators used in the expression. We require that every variable used in an expression in a mode be declared in that mode. The only allowed type conversion is from `int` to `real`.

Furthermore, an expression used in an assignment statement within an update block must have the same type as the variable on the left-hand side of the assignment statement. Expressions in invariants must have type `bool`. Controlled entities and expressions in constraints must be of type `real`. In addition, controlled entities must be declared `analog`.

We want to avoid a pathological situation when the mode takes a transition to a submode, but control cannot enter the submode because none of the entry transitions from the chosen entry point of the submode are enabled. Thus, we require that it is always possible to enter or exit a mode. This means that for every entry and exit point of each mode (except default entry and exit points) there is at least one enabled transition leading from that point. In other words, the disjunction of the boolean expressions in the guards of all transitions leading from a control point must be `true`. Default control points are used in modeling interrupts and are supplied with default entry and exit transitions which the user does not have to specify explicitly. Thus, default control points automatically satisfy this requirement.

4.1.5 Agents

Agents describe how concurrent system components communicate with each other. That is, unlike modes that describe system behavior, the hierarchy of agents describes system architecture. Agents communicate with each other through shared variables and channels, which form an external interface of an agent. In addition, an agent can have private variables and channels that can be used only for communication between subcomponents of the agent.

Structure of an agent

An agent is consists of a set of variable declarations, a static initialization block, and a set of instantiations. An agent is introduced by the agent header with the keyword `agent`, a name and a set of parameters. Parameters and variable declarations are exactly the same as in the case of a mode declaration. The initialization blocks contains a sequence of assignments to the variables of the block. Assignments are expressions composed of constants, parameters, and values of variables initialized earlier in the same block. Initialization blocks of a system of agents are executed once before the execution of the system. They are not a part of system behavior, but but specify the initial state for the system. The order of execution of initialization blocks in different agents is undetermined.

Instantiations. Instantiation statements define components of an agent. We distinguish between two kinds of agents: primitive and composite agents. Primitive and composite agents have different instantiation statements. We discuss each kind of agents in turn.

Primitive agents. An example of a primitive agent is given below.

```
agent A_M( int p1, real p2 ) {
    // variable declarations
    private analog real clock;
    // initialization
    init { clock = p2; }
    // instantiation statements
    mode top = M(p1, p2);
}
```

A primitive agent is a single component that does not have any concurrent behavior inside. Its behavior is given by a single mode. The variables of the agent are, essentially, the variables of the mode, except that the agent may declare some of them private. Primitive agents have only one agent statement, which is an instantiation of its mode. The mode instantiation is done in the same way as in mode declarations. In the example, the agent *A_M* encapsulates the mode *M* that was used to illustrate mode declarations. Note that the variable *clock* of *M* is made private in the agent, but variable *skew* remains public. There is no need to re-declare variables of the mode in the agent, unless the are made private or used in the initialization block.

Composite agents. Composite agents consist of other agents running in parallel. The statements of a composite agent are agent instantiations. Agent instantiations give values to parameters in the same way mode instantiations do. In additions, variables of an instantiated agent may be renamed either to avoid conflict between variables of the same name in different agents or to enforce communication between agents.

Shared variables. Sub-agents of a composite agent communicate through shared variables. Global variables of agents are shared based on their names. That is, variables of the same name in different agents denote the same variable shared among these agents. Shared variables are required to be declared identically among all agents that share them.

```
agent Acomp( ) {
    // variable declarations
    private analog real clock;
    // agent statements
    agent runClock = A_M(10, 0) [skew := clock];
    agent readClock = Aother();
}
```

In the example above, an instance of agent *A_M* is run in parallel with an agent *readClock*, communicating through a variable *clock*. Note that *A_M* also has a variable *clock*, but it is private and thus cannot be used for communication. The only global variable of *A_M* is *skew*, which behaves like a skewed clock (see the description of mode *M* in Section 4.1.4). In the instantiation of *A_M*, *skew* is renamed to *clock* so that the two agents can communicate through this variable..

4.1.6 External objects

External objects in CHARON models are used for communication with the environment of the system. In this way, programmers gain access to libraries of commonly used functions and other facilities that are not present in CHARON itself. We use Java naming conventions to describe external objects. In the case of CHARON simulator, which is implemented in Java, the referenced objects are assumed to exist in the virtual machine that runs the simulator and thus are accessed directly. In other cases, the implementation will map the CHARON external objects to the objects of the runtime environment.

Two kinds of external objects are available: variables and functions. Both are identified by a dot-separated sequence of names extended, in the case of functions, with a list of parameter types. In the CHARON simulator, external variables and functions are understood as static fields and methods of classes known to the simulator. These classes can be either from standard Java packages, or from the CHARON simulator package. An example of commonly used external functions are trigonometric functions from the `Math` class. Such a declaration in a CHARON model will be

```
real java.lang.Math.cos(real);
```

4.2 Simulation of CHARON Models

In this section, the informal semantics for CHARON models by means of a simple global simulator for CHARONis presented.

Simulation of CHARON model comprises three phases, initialization phase, discrete update phase, and continuous update phase. After initializing agents, simulator executes either discrete update phase or continuous update phase. In other words, simulation of CHARON model is *initialization phase* (*discrete update phase* | *continuous update phase*)$*$.

4.2.1 Example: Bi-rate clock

We describe the simulator using an example presented below. This example demonstrates an agent with two submodes. Agent biRate represents a clock runs at two different rates, *r1* and *r2*, where *r1* is lower than *r2*. The clock goes slow at the rate of *r1* in *lowRate* and fast in *highRate* mode. Every ten time units, the agent changes its clock rates between *r1* and *r2* by turns. Invariant *invClock* ensures that the value of clock is always non-negative.

```
agent biRate() {
    mode top = biRateTop(1.0, 2.0)
}

mode biRateTop(real r1, real r2) {
    private analog real clock;

    inv invClock { clock >= 0.0 }

    mode lowRate = updateTime(r1);
```

```
    mode highRate = updateTime(r2);

    trans toSubMode from default to lowRate when true
        do { clock = 0.0 }
    trans toHigh from lowRate to highRate when (clock % 20.0 == 10.0)
        do { }
    trans toLow from highRate to lowRate when (clock % 20.0 == 0.0)
        do { }
}

mode updateTime(real a) {
    readWrite analog real clock;

    diff dClock { d(clock) == a }
}
```

4.2.2 Initialization

The agents are initialized by providing initial values for variables and by configuring active modes. With initialization done, the active modes of an agent are the modes included in the path from the top mode to a leaf mode. For example, in bi-rate clock, *top* mode and *lowRate* mode are the active modes and the initial value of *clock* is 0.0 after the initialization phase is finished.

4.2.3 Discrete update

The discrete update phase executes a sequence of enabled transitions. The simulator picks an agent randomly and execute a transition enabled in a submode of this agent. The transition is enabled if it is in an active mode and its associated guard is satisfied. When a transition happens, the statements in its update block are executed, updating the variables of the block. Since a transition leaves one submode and enters another one, the set of active modes is updated after the transition is executed. After this, another transition enabled in the same agent is chosen for execution. This phase can end when control reaches a default exit point. The logical time of simulator does not change during a discrete update.

In the example, transition *toHigh* and *toLow* are enabled if clock is 10, 30, 50, ... and 20, 40, 60, ..., respectively. If there are multiple enabled transitions, one of them is non-deterministically selected for execution.

4.2.4 Continuous update

Continuous update phase is a time flow of an arbitrary duration that does not violate constraints of any mode. Constraints of a mode consist of an invariant and differential and algebraic constraints. In this phase, all the active modes of agents are executed together. The duration of a continuous update is chosen before it starts. However, a continuous update is terminated before the chosen interval elapses if a violation of a constraint is detected. It also may be terminated early if a transition becomes enabled in one of the agents. In this case, however, the decision to terminate the flow is left to the simulator.

The discrete update phase may preempt the continuous update phase, if a transition gets enabled. For example, if active modes are *top* and *lowRate* mode, the variable *clock* will get updated at the rate of *r1* as long as invariant *invClock* is not violated. The transition *toHigh*, however, may get executed if *clock* % 20.0 is 10.0.

Bibliography

[1] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling of embedded systems. In *First Workshop on Embedded Software*, number 2211 in Lecture Notes in Computer Science, Oct. 2001.

[2] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings of Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 33–48. Springer-Verlag, Mar. 2001.

[3] X. Liu, Y. Xiong, and E. A. Lee. The Ptolemy II framework for visual languages. In *poster paper, Symposium on Visual Languages and Formal Methods*, Sept. 2001.

Appendix A

Text of the Leaky Tank model

```
extern real java.lang.Math.random();
extern real java.lang.Math.abs(real);

// ---------------------------------------------------------------------------
// a leaky tank controlled by a pump

agent LeakyTank() {
  private analog   real level, flow;

  agent tank = LTank( )  [ inflow := flow ]
  agent pump = Pump( 5, 10 )
}

// ---------------------------------------------------------------------------
// the tank agent with a hidden leak

agent LTank() {
  private analog real leak;

  agent tank = Tank();
  agent hole = Hole();

}

// ---------------------------------------------------------------------------
// the hole agent: leak changes randomly every so often
agent Hole() {
  write analog real leak;

  init { leak = 1; }

  mode top = HoleMode();
}

mode HoleMode() {
  write analog real leak;
  private analog real clock;
  private discrete real bound;
```

```
  mode dummy = empty();

  trans start  from default to dummy when true do { clock = 0; bound = 1; }
  trans change from dummy to dummy when clock >= bound
   do { bound = java.lang.Math.random()+0.5;
leak = java.lang.Math.random()*5;
clock = 0; }
  diff { d(clock) == 1; d(leak) == 0 }
}

// ---------------------------------------------------------------------------
// the tank agent: water level is controlled by the difference between in
//                 and out flows
agent Tank() {
  write   analog real level;

  init { level = 6; }

  mode top = TankMode( );

}

mode TankMode() {
  read    analog real inflow;
  write   analog real level;
  read  analog real leak;

  diff { d(level) == inflow-leak }

  assert { 0 <= level and level <= 15 }
}

// ---------------------------------------------------------------------------
// the pump agent

agent Pump( int low, int high ) {
  write analog real flow;
  read  analog real level;

  init { flow = 0; }

  mode top = PumpMode( low, high );
}

mode PumpMode( int low, int high ) {
  private analog real clock;
  private discrete real rate;
  private discrete int  hist;
  write analog real flow;
  read  analog real level;

  mode m = Maintain( 0.1, low, high );
  mode c = Compute();
```

```
  trans from default to m       when true      do { clock = 0; rate = 0; hist = 0 }
  trans from m       to c.start when clock > 1 do { }
  trans from c.ret   to m       when true     do { clock = 0; }

  diff { d(clock) == 1; }
}

mode Maintain( real tolerance, int low, int high ) {
  write analog real flow;
  read    analog real level;
  write   discrete int  hist;
  read    discrete real rate;

  mode on =      SteadyMode( tolerance, 0, high );
  mode off =     SteadyMode( 15, low, 10000000 );
  mode adjust =  TransientMode( tolerance, high );
  mode turnOff = TurnOffMode( low );

  trans initM1  from default to on      when hist == 0     do {}
  trans initM2  from default to turnOff when hist == 1     do {}
  trans initM3  from default to off     when hist == 2     do {}
  trans initM4  from default to adjust  when hist == 3     do {}
  trans reqOff  from on      to turnOff when level > high  do { hist = 1 }
  trans doneOff from turnOff to off     when flow <= 0     do { hist = 2 }
  trans reqOn   from off     to adjust  when level < low   do { hist = 3 }
  trans doneOn  from adjust  to on
         when java.lang.Math.abs(rate-flow) <= tolerance do { hist = 0 }
  trans reqAdj  from on      to adjust
          when java.lang.Math.abs(rate-flow) > tolerance do { hist = 3 }
  trans quitOff from turnOff to adjust  when level <= low  do { hist = 3 }
  trans reqOff1 from adjust  to turnOff when level > high  do { hist = 1 }
}

mode SteadyMode( real tolerance, int low, int high ) {
  write analog real flow;
  read  analog real level;
  read  discrete real rate;

  diff { d(flow) == 0 }

  inv { level >= low && level <= high &&
        java.lang.Math.abs(rate-flow) <= tolerance  }
}

// adjust a pump to the desired rate;
// the rate can be changed during a adjustment
mode TransientMode( real tolerance, int high ) {
  write analog real flow;
  read  analog real level;
  read  discrete real rate;

  diff { d(flow) == 5*(rate-flow) }
```

```
  inv { java.lang.Math.abs(rate-flow) > tolerance && level <= high }
}

// turn off the pump: works like the transient mode with the desired rate of 0
// dangerously low levels will abort the turn off
mode TurnOffMode( int low ) {
  write analog real flow;
  read  analog real level;

  diff { d(flow) == -5*flow }

  inv { flow >= 0 && level >= low }
}

mode Compute() {
  entry start;
  exit  ret;

  write discrete real   rate;
  read  analog   real   level;
  private discrete real past, delta;

  mode pt0 = point();
  mode pt1 = point();

  trans comp0 from start to pt0 when true         do { delta = level - past }
  trans comp1 from pt0   to pt1 when delta <= -20 do { rate = 20 }
  trans comp2 from pt0   to pt1 when delta >= 0  do { rate = 0 }
  trans comp3 from pt0   to pt1 when delta < 0 && delta > -20
       do { rate = -delta }
  trans comp4 from pt1   to ret when true do { past = level; }
}

// an empty mode: cannot do anything except pass time
mode empty() {
}

// a "computational node": cannot spend time, no dynamics
mode point() {
  mode d = empty();

  inv { false }
}
```