CHARON Language Manual

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, interfaces, and scoping rules for variables similar to 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 start by presenting a detailed description of CHARON syntax in Section 2. The syntax described here corresponds to CHARON grammar version 0.6. Then, in Section 3, we describe the informal semantics of CHARON by means of a high-level description of a CHARON simulator.

2 The CHARON language

This language presents the detailed syntax of the CHARON language, along with the consistency requirements for CHARON programs that ensure that their semantics is well-defined.

2.1 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 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`, `bool`, `channel`, `cold`, `default`, `diff`, `discrete`, `do`, `entry`, `exit`, `extern`, `false`, `from`, `hide`, `hot`, `in`, `init`, `int`, `inv`, `isempty`, `macro`, `mode`, `of`, `or`, `private`, `read`, `readwrite`, `real`, `receive`, `send`, `struc`, `to`, `trans`, `true`, `when`, `while`.

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

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

Program structure. A CHARON program 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 2.4, 2.5 and 2.6, 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 to conclude variable declarations and statements, such as assignment statements and instantiations of modes and agents. Without going into too much detail, semicolons can be used after every construct, unless it ends in a right brace (}). When in doubt, it is always safe to omit a semicolon. The only useful purpose they serve in CHARON programs is to let C and Java programmers feel more comfortable.

2.2 Variables and channels.

Variables in CHARON are assigned types. Types are a subset of Java types and are either primitive or user-defined. Primitive types are integer (`int`), real (`real`), and boolean (`boolean`). User defined types are dot-separated sequences of names.

In addition, variables are classified as either discrete of 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 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 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, readwrite,` and `private`. An access specifier cannot be omitted from a variable or channel declaration.

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 before it 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, one of the following happens. Either the new value is dropped and the data in the channel queue stays unchanged. This is referred to as the "cold" policy. 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.

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

⟨access specifier⟩ ⟨update specifier⟩ type ⟨list of names⟩

or

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

All of the following are valid declarations:

`read discrete int` *i1, i2*;

`readwrite analog` robot.Position p;

`write channel`[25] `of real` *ch_out*

`private int` *distance*

2.3 Expressions

Expressions in CHARON programs 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 `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.

2.4 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 the mode, control is passed along the transitions of a mode from one submode to another along the transitions. Such discrete execution obeys the guards that enable transitions to be taken and updates the variables of the mode when a transition is taken. 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 mode and the active submode.

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 all 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 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-only mode variables. Mode parameters should not be thought as parameters of a function in programming languages. Rather, they are constants that are used to instantiate the mode before it can be used. In the example, we have two parameters, *p1* of type `int` and *p2* of type `real`.

Control points 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 2.2. In our example, we have two variables in mode *M*, both of type real. One of these variables is private, that is, not visible from the outside. The other variables is written by the mode *M* and can be read by other modes and agents.

Mode instantiation. A mode can have a number of submodes. Each submode is a mode instance, created by the 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 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, we have both differential and algebraic constraints.

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

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` $\{\ \mathit{clock} < \mathit{p1}\ \}$. 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 not more than 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 channel 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 the 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, we have 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 s updated here in a discrete fashion.

A more general form of a transition may involve multiple destination points, each with its own guard, and a *group guard*. 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); }
    ...
```

We assume that if the group guard is satisfied, then at least one of the other guards in the transition will be satisfied as well.

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. 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. A default control point can also be identified by the keyword `default`. In the example above, entry point *in* can also be addressed as *M.in*. Other examples of control points are: $M1$.`default` is a default control point of $M1$; M, M.`default`, or simply `default` all refer to a 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 an entry point if it belongs to a mode or an exit point if it belongs to 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 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 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 union of the guards of all transitions leading from a control point must be `true`.

2.5 Agents

Agents describe how concurrent system components communicate with each other. That is, unlike modes that describe the system behavior, the hierarchy of agents describes the 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.

Agent statements are instantiations of the components of the agent. We distinguish between two kinds of agents: primitive and composite agents. Examples of an atomic and a composite agent are given below.

```
agent A_M( int p1, real p2 ) {
    // variable declarations
    private analog real clock;
    // initialization
    init { clock = p2; }
    // agent 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 private in the agent, but variable *skew* is public. There is no need to re-declare variables of of the mode in the agent, unless their declarations change.

Composite agents consist of other agents running in parallel. Concurrent agents communicate through shared variables. Global variables of the same name in all agents are shared. 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.

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

In the example, 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. Its only global variable is *skew*, which behaves like a skewed clock (see the description of mode M in Section 2.4). In the instantiation of *A_M*, *skew* is renamed to *clock* so that the two agents can communicate.

2.6 External objects

External objects in CHARON programs 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 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 program will be

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

3 Simulation of CHARON program

In this section, we present the informal semantics for CHARON programs by means of a simple global simulator for CHARON.

Simulation of CHARON program 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 program is *initialization phase* (*discrete update phase* | *continuous update phase*)∗.

3.1 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*. *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 = dummy(r1);
    mode highRate = dummy(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 dummy(real a) {
    readWrite analog real clock;

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

3.2 Initialization

We initialize agents by providing initial value for variables and 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 initialization phase is finished.

3.3 Discrete update

Discrete update phase is a sequence of executing enabled transitions. We pick 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.

3.4 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.