Modular Specification of Hybrid Systems in CHARON

Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee

Department of Computer and Information Science, University of Pennsylvania,
Philadelphia PA 19104-6389, USA,
alur,grosu,yehur,kumar,lee@cis.upenn.edu,
URL: http://www.cis.upenn.edu/~alur,grosu,yehur,kumar,lee

Abstract. We propose a language, called CHARON, for modular specification of interacting hybrid systems. For hierarchical description of the system architecture, CHARON supports building complex agents via the operations of instantiation, hiding, and parallel composition. For hierarchical description of the behavior of atomic components, CHARON supports building complex modes via the operations of instantiation, scoping, and encapsulation. Features such as weak preemption, history retention, and externally defined Java functions, facilitate the description of complex discrete behavior. Continuous behavior can be specified using differential as well as algebraic constraints, and invariants restricting the flow spaces, all of which can be declared at various levels of the hierarchy. The modular structure of the language is not merely syntactic, but can be exploited during analysis. We illustrate this aspect by presenting a scheme for modular simulation in which each mode can be compiled solely based on the locally declared information to execute its discrete and continuous updates, and furthermore, submodes can integrate at a finer time scale than the enclosing modes.

1 Introduction

A hybrid system typically consists of a collection of digital programs that interact with each other and with an analog environment. The design and implementation of hybrid systems remains a challenging task. We believe that availability of a specialized design language for hybrid systems will aid the developers significantly and lead to opportunities for greater design automation. Traditional tools for modeling and simulation of dynamical systems, such as MATLAB (see www.mathworks.com), provide little support for modular specifications. On the other hand, modern software design languages, such as STATECHARTS [Har87] and UML [BJR97], provide no support for describing continuous behavior. In this paper, we introduce a language, called CHARON, for hierarchic specification of interacting hybrid systems. The design of our language was guided by two concerns. First, the language should support state-of-the-art modeling concepts such as encapsulation, reuse, preemption, and hierarchy. Second, it should be possible to give a modular formal semantics to the language which can be exploited during simulation, verification, and code generation.

In Charon, a system is described as a collection of agents communicating via shared variables, and the behavior of each agent is specified by a hierarchical state machine. Key features of Charon are summarized below.

Architectural hierarchy. 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.

Behavior hierarchy. The building block for describing flow of control inside an atomic agent is a *mode*. A mode is basically a hierarchical state machine, that is, a mode can have submodes and transitions connecting them. Variables can be declared locally inside any mode with standard scoping rules for visibility. Modes can be connected to each other only via well-defined entry and exit points. We allow *sharing* of modes so that the same mode definition can be instantiated in multiple contexts. Finally, to support *exceptions*, the language allows group transitions from default exit points that are applicable to all enclosing modes, and to support *history retention*, the language allows default entry transitions that restore the local state within a mode from the most recent exit.

Discrete updates. Discrete updates are specified by *guarded actions* labeling transitions connecting the modes. We assume *interleaving* semantics for concurrency (i.e., only one atomic agent is executed in a discrete round), *run-to-completion* semantics for individual agents (i.e., once an agent is chosen for discrete update, it keeps executing its transitions as long as there are enabled ones), and higher priorities for inner modes (i.e., group transitions from the default exit of a mode are examined only when there are no enabled transitions inside).

Continuous updates. Some of the variables in Charon can be declared *analog*, and they flow continuously during continuous updates that model passage of time. The evolution of analog variables can be constrained in three ways: *differential* constraints (e.g. by equations such as $\dot{x} = f(x, u)$), *algebraic* constraints (e.g. by equations such as $y = g(x, u)$), and *invariants* (e.g. $|x - y| \leq \varepsilon$) which limit the allowed durations of flows. Such constraints can be declared at different levels of the mode hierarchy.

It should be noted that Charon is a *modeling language*: it supports nondeterminism for both discrete and continuous updates, it is suitable for describing the system as well as the assumptions about the environment in which the system is supposed to operate, and for describing the same system at different levels of abstraction. The language constructs primarily facilitate the description of control flow, but it also supports calls to externally defined Java functions which can be used to write complex data manipulations.

After introducing the language in the next two sections, we proceed to illustrate how to exploit the modular structure during simulation. Since modes are hierarchical, multiple modes within an atomic agent can be active simultaneously, and a large number of transitions may be applicable in a given state.

In our modular scheme for discrete updates, each mode gets compiled into a function which gets control at one of its entry points along with an input global state, and returns the control at one of its exit points together with a modified global state. Such a modular scheme is possible since Charon modes have explicit entry and exit points including the default ones, and inner transitions have higher priorities over the outer ones.

Introducing modularity in simulation of time rounds is more challenging. Since time is global, update of analog variables of all agents must be synchronized. Furthermore, within a single agent multiple modes are active, and the constraints on continuous update may be defined at any level of the hierarchy. This implies that simulating a flow requires solving constraints of all active modes of all agents simultaneously. In a modular scheme, we wish to compile each mode independently of the other.

Concurrency. To handle concurrency, we propose a scheme for distributed simulation in which each agent has its own local clock. The scheme ensures that the differences among local clocks are bounded. In any time round, we update the analog variables of only one of the agents keeping the state of the remaining agents unchanged.

Hierarchy. Each mode is responsible for integrating the variables whose update laws are defined locally, at a time scale of its own choice based on the local control laws and the invariants. A mode M is invoked from higher level with an input state, a bound δ on integration time, and an invariant constraint on the local variables of M. The integration within M assumes that the variables whose update laws are defined outside M stay unchanged. It can choose to integrate at time intervals shorter than δ, and can use integration routines of its submodes as black-boxes.

In summary, instead of solving the entire set of constraints simultaneously, the modular scheme computes the approximate solutions by layering the constraints as dictated by the modular specification.

Related work. Early formal models for hybrid systems include phase transition systems [MMP91] and hybrid automata [ACH$^+$95]. There has been a lot of research concerning analysis of hybrid automata leading to the model checker HyTech [AHH96,HHW95]. Models such as hybrid I/O automata [LSVW96] and hybrid modules [AH97] allow compositional treatment of concurrent hybrid behaviors. None of these models admit hierarchical specifications.

The notion of hierarchical state machines was introduced in Statecharts [Har87], and is present in many software design paradigms such as Uml [BJR97]. Our treatment of hierarchy is closest to hierarchical reactive modules [AG00] which shows how to define a modular semantics for hierarchical (discrete) modes.

The languages Shift [DGS97] and HyCharts [GSB98] allow hierarchal specifications of hybrid behavior, and Stateflow (see `www.mathworks.com`) allows hierarchal specifications of dynamic behavior. However, modular simulation has not been a concern in the design of these languages. Furthermore, Charon supports new features such as preemption and reuse that are important from a programming perspective.

2 Language Overview

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 composed agents starting from the most primitive ones called atomic agents. Information flow inside a composed agent may be hidden to the outside world. The grouping of agents into composed agents gives the architecture of the hybrid system. A composed 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. It may be also regarded as a behavioral pattern that may be instantiated.

For example, at a lower level, one may consider for a robot the modes walkForward, walkLeft, walkRight and walkBackward. At a higher level one may consider the modes avoidObstacle and trackWall. avoiding obstacles.

Note that an atomic agent is nothing but a hierarchical mode. Its variables and behavior are completely determined by the mode. Moreover, a hierarchical agent is nothing but a set of hierarchical modes with local variables determined by the agent hierarchy. So why do we distinguish between modes and agents? The answer is that encapsulating modes inside agents prevents parallel composition inside modes, i.e., modes are entities composed in a purely hierarchic way.

2.1 Variables

Discrete and analog variables. A hybrid agent has a finite set of typed variables denoted $A.V$. Some of these variables are updated in a discrete fashion and the others change in an analog fashion when time elapses. Accordingly, the set $A.V$ is partitioned in two sets, the set $A.dscV$ of *discrete variables* and the set $A.anaV$ of *analog variables.*

Differential and algebraic variables. In control theory it is common to compute the values of the analog variables $A.anaV$ by using algebraic and differential equations. For example, $\dot{x}{=}f(x,u)$ is a differential equation whereas $y{=}g(x,u)$ is an algebraic equation. Regarding f and g as functional blocks and x, y, u as wires, it is easy to see that the wire x is a feedback loop of f. As a consequence, the current value of the output x of f depends on the previous (infinitisimal) value of x. In contrast, the current value of the output y of g depends only on the current values of the inputs x and u. Hence, an algebraic equation is very similar to a combinational circuit whereas a differential equation is similar to a sequential circuit. In CHARON we generalize algebraic equations also to inequalities. We call the differential equations and algebraic equations generically as *constraints*. The variables defined by algebraic constraints are called *algebraic variables* and the variables defined by differential constraints are called *differential variables*. Hence, $A.anaV = A.\mathit{diffV} \cup A.algV$. We insist that $A.\mathit{diffV} \cap A.algV = \emptyset$. Note that hybrid automata don't make any distinction between these two kinds of variables.

Permitted read/write accesses. The variables $A.V$ of an agent A are classified according to their visibility and update permissions into three sets: the set $A.lclV$ of *local variables* that cannot be read or written by other agents, the set $A.wrtV$ of *write variables* that are written by A, and can be read by other agents, and the set $A.readV$ of *read variables* that are read by A, and may be written by other agents. The sets $A.readV$ and $A.wrtV$ need not be disjoint. Similarly, the set of local variables $A.lclV$ may be both read and written. The set of read and write variables $A.gblV = A.readV \cup A.wrtV$ is used for communication and it is called the set of *global variables*. The set $A.updV = A.wrtV \cup A.lclV$ of write and local variables is called the set of *updated variables*. Hence, our communication model is that of asynchronous communication over shared variables. This model is a very general and allows to define channels as a special case.

States and actions. Given a set V of typed variables, a *state* over V is a function mapping variables to their values. Given two sets V and W of variables, an *action* from V to W is a binary relation between the states over V and the states over W. In CHARON specifications, an action consists of an action *guard* over V and an action *body* from V to W. We say that an action is enabled (disabled) at a state s if its guard is true (false) at that state.

2.2 Hierarchical Modes

Hierarchy. A *mode* in CHARON has a very refined control structure, given by a hierarchical, hybrid state machine. It basically consists of a set of *submode references* connected by transitions such that at each moment of time only one of the submode references is active. A submode reference has associated again a mode and we require that the modes form an *acyclic* graph with respect to this association. By using modes and mode references several references may share the same mode. This is highly desirable because modes in a definition are never simultaneously active. A mode resembles an *or* state in STATECHARTS, but it has more powerful structuring mechanisms.

Variables. A mode has global as well as local variables. Global variables are used to share data with the mode's environment, and are partitioned into the set *readV* of *read variables* and a set *wrtV* of *write variables*. The set $gblV = readV \cup wrtV$ is called the set of *global variables*. The set of *local variables* *lclV* of a mode is accessible only by its transitions and submodes. Thus, the scoping

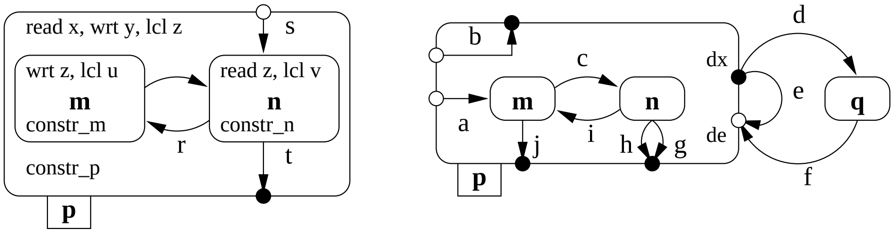

Fig. 1. Scoping rules and transition types

rules for variables are as in standard structured programming languages. For example, in Figure 1 left, the transitions of the mode p (like r, s and t) may refer only to the variables x, y and z. These variables are global to the modes referred to by m and n. However, the variables local in the mode referred to by m may not be used in the mode referred to by n. For example, in Figure 1 left, the variable z may be accessed both in m and n but the variables u and v are private to m and n, respectively .

Dynamics. A mode has an associated set of *constraints*. These include differential equations, algebraic equations and *invariants* that are differential and algebraic inequalities. The constraints define the flows of the mode, i.e., the way analog variables are updated while the agent is in this mode. The invariants define conditions that have to be satisfied by the variables in this mode, i.e., they define allowed durations. The scoping rules also apply for these constraints. For example, in Figure 1, constr_p may only refer to x, y and z and constr_m may refer only to z and u. For each differential and algebraic variable updated by a mode we require that the variable is either updated by the mode itself or it is updated by all submodes of this mode. For example, in Figure 1, the local variable z is either updated by a differential constraint in the mode p or by differential constraints in both submodes m and n.

Interfaces. To obtain a modular language, we require the modes to have well defined *control points* classified into entry points (marked as white bullets) and exit points (marked as black bullets). The transitions connect the control points of a mode and its submode references to each other. For example, in Figure 1 right, a is an *entry* transition, g, h and j are *exit* transitions, b is an *entry/exit* transition and c and i are *internal* transitions. Between these transitions there is a subtle difference. Entry transitions initialize the local variables by reading only from the global variables. Exit transitions forget the values of the local variables by writing only to the global variables. It is only the internal transitions that may both read and write the local variables.

Preemption. To model preemption we use the special default exit point *dx*. A transition starting from the default exit point of a mode is called a *group transition*. It may be taken whenever the control is inside the mode and no internal transition is enabled. For example, in Figure 1 right, the group transition d is taken if it is enabled and all the transitions c, g, h, i, and j are disabled. Hence, inner transitions have a higher priority than the group transitions, i.e., we use *weak preemption* (like the weak kill in Unix, versus the strong kill -9). This definition of priorities allows us to define in Section 4 a modular simulation.

History. To allow history retention, we use the special default entry point *de*. A transition entering the default entry point of a mode restores the values of all local variables along with the position of the control (a transition may enter a default entry of a mode only if the mode was left along its default exit). For example, both transitions `e` and `f` in Figure 1 right, enter the default entry point. The transition `e` is called a *self* group transition. A self transition (like `e`) or more generally a self loop like `d, q, f` may be understood as an interrupt handling routine. While a self loop may be arbitrary complex, a self transition may do simple things like counting the number of occurrences of an event (e.g., clock events).

The *set of modes* in a CHARON specification is supposed to be globally accessible. Moreover, since a mode may refer to other modes we require that referencing forms an *acyclic graph*.

Leaf and top level modes. A *leaf mode* is a mode with no submodes and a default *identity* transition from its default entry point *de* to its default exit point *dx*. A *top-level* mode is a mode M with a single explicit entry point e and no exit points.

Mode operations. The mode definition can be viewed as an *encapsulation* operator over its submodes, and thus, modes are constructed from leaf-modes using encapsulation repeatedly in a non-recursive manner. Since encapsulation may partially overlap, the mode structure looks like a directed acyclic graph, rather than a tree, and this can be exploited during analysis.

2.3 Hierarchical Agents

An atomic agent is basically a top level mode whose global variables are used for communication with other agents. As we already mentioned, atomic agents may be composed to form composed agents and communication inside composed agents may be hidden. Intuitively, composition of atomic agents is the union of their modes and hiding is a declaration of local variables. To make the operations over agents closed under composition and hiding, we define an agent as follows.

Definition 1. *(Agent) An* agent P *is a tuple consisting of*

Modes. *A set of top-level modes* M*.*

Local variables. *A set* $lclV \subseteq \cup_{m \in M} m.gblV$ *of local variables.*

Global variables. *A set* $gblV = (\cup_{m \in M} m.gblV) \setminus lclV$ *of globals variables.*

Definition 2. *(Composition) If A and B are two agents, then the* composition $A \| B$ *is the agent with the set* $lclV = A.lclV \cup B.lclV$ *of local variables, the set* $wrtV = A.wrtV \cup B.wrtV$ *of write variables, the set* $readV = A.readV \cup B.readV$ *of read variables and the set* $M = A.M \cup B.M$ *of top level modes.*

Definition 3. *(Variable Renaming) Let A be an agent, $x \in A.gblV$ a global variable of the agent and $y \notin A.V$ a variable of the same type as x but not contained in A. Then the* renaming $A[x := y]$ *is the agent obtained by consistently renaming x by y in $A.V$ and in all modes $m \in A.M$.*

Definition 4. *(Variable Hiding) Let A be an agent, $x \in A.gblV$ a global variable of the agent. Then the* variable hiding **hide** x **in** A *is the agent obtained by replacing $A.gblV$ with $A.gblV \setminus \{x\}$ and $A.lclV$ with $A.lclV \cup \{x\}$.*

3 Global Semantics

One alternative in giving a semantics to a hierarchical system is to consider hierarchy as just a convenient syntactic abbreviation. This reduces the semantic definition to two considerably easier subproblems: a) show how to construct a flat system out of the hierarchical one and b) give a semantics to the flat system.

3.1 The Flattening Operation

Given a mode definition, the flattening operation recursively eliminates the submode references as follows: a) take for each reference m the associated definition, b) prefix all elements of the mode definition by m, c) continue recursively until all references point to a leaf mode definition. The set of elements obtained this way are taken as the elements of the flat mode.

As a consequence of flattening, all elements of the resulting mode are prefixed with a path m_1:m_2:...:m_k from the root mode reference m_1 down to the containing mode reference m_k of the original hierarchical mode. For example, a control point c has now the form m_1:m_2:...:m_k:c. The set of local variables $flat(M).lclV$ of the flattened mode $flat(M)$ is the transitive closure of the local variables of M and the local variables of its submodes.

In the semantic definitions of the next section we model paths by *stacks*. Textually, we write stacks with the elements separated by colons and with the topmost element on the left. For example `s = a:b:s'` is the stack `s` with the top element `a`, the second element `b` and the rest of the stack `s'`. To show how stacks evolve in a pictorial way we use pattern matching. For example when we write `if ((as = a:b:as') & (bs = c:bs')) (as,bs) = (c:as', a:b:bs')` we mean that if the current value of the stack `as` has topmost elements `a` and `b` and the current value of the stack `bs` has the topmost element `c` then the next value of `as` has discarded `a` and `b` and pushed `c`, and the next value of `bs` has discarded `c` and pushed `a` and `b`.

3.2 Update Rounds

In an update round, the semantic function nondeterministically chooses one of the modes of the resulting flat agent and executes the discrete update on that mode. Using a pseudo-code like notation this can be described as shown below.

```
State updateRound (Agent a, State s){
   return forany (m in subModes(a)) discreteUpdate(m, s); }
```

The discrete update of a mode is a sequence of enabled implicit and explicit transitions starting at the default entry point of the mode and ending at the default exit point of the mode. The algorithm for generating this sequences is given below. In the first step it uses the global history variable `hs`, that is itself a stack, to execute a series of default entry transitions down to the last control point where the explicit execution got stuck, i.e., where all the explicit transitions were disabled. A default entry transition restores the saved submode and point by popping them from the history stack and pushing them on the control stack `ct`.

```
State discreteUpdate (Mode m, State s) {
   Stack ct = de:m:[]; State st = s;       //put de and m
   while (ct != dx:m:[]) {                 //while dx not reached
      while (ct = de:ct')                  //while de is the top point
         if (st.hs = pt:md:hs')
         (ct, st.hs) = (pt:md:ct', hs');   //default entry transition
         else ct = dx:ct';     //default identity transition for leaf mode
      while (enabledFanOut(ct, st) != {})
         (ct, st) = forany (t in enabledFanOut(ct, st)) t(ct, st);
      let (ct = pt:md:ct') in
         if (pt != de)
         (ct, st.hs) = (dx:ct', (pt=dx?de|pt):md:st.hs) }
   return st; }
```

If the history stack hs is empty and the top point on the control stack ct is the default entry point de then a leaf mode has been reached and the identity transition of the leaf mode is executed.

In the second step, the algorithm executes a sequence of explicit, enabled transitions starting at the control point obtained in the previous step and ending at the control point where all the explicit transitions are disabled. The enabled fanout of a mode reference is the set of enabled transitions in the associated mode definition, with source point pt and with source state st.

In the third step, the algorithm executes an implicit exit transition provided that the last transition was not a self group transition (in this case, the top point pt is equal to de). The default exit transition saves the relative value of the control point from the previous step on the top of the history stack and passes the control to the default exit of the parent mode. Note that, if the top point on the control stack ct was the default exit point dx, then the exit transition saves on the history stack hs the default entry point de. This assures that in the next step, the deepest point is tried first.

Since the top of the control stack is dx and not de, the first step is skipped when control is passed up to the parent mode. The second step in this case amounts to executing a *group transition* if any enabled transition exists. If this is not the case, the control is passed in the third step up again to the enclosing parent mode and so on up to the top mode. If any of the group transitions is enabled, then executing this transition (and possibly other), may return the control to the default entry point de of the mode, and the algorithm proceeds by skipping the third step and executing all the default entry transitions.

3.3 Time Rounds

In a time round, for a given state s_1, the semantic function executes for a time interval d, and produces a new state $s_2 = s(d)$, where s is any flow that is a solution of the active set of control constraints, not violating the current set of invariants and such that $s(0) = s_1$. The semantic function is shown below, where the type Constraints is assumed to contain a set of algebraic constraints, a set of differential constraints and a set of invariants.

```
State timeRound (Agent a, State s) {
   Constraints c = agentConstraints(a, s);
   return forany ((f, d) in solution(c, s)) f(d); }
```

The set of active constraints for an agent is the union of the active constraints of each mode in the agent.

```
Constraints agentConstraints (Agent a, State s) {
   Constraints ac = {};
   forall (m in modes(a))
      ac = ac ∪ modeConstraints(m, s);
   return ac; }
```

For each mode, the set of active constraints is easily recovered form the history variable.

```
Constraints modeConstraints (Mode m, State s) {
   Constraints mc = getConstraints(m); Stack hs = s.hs
   while (hs = pt:md:hs') {
      mc = mc ∪ getConstraints(md);
      hs = hs'; }
   return mc; }
```

Hence, in a global semantics, the flows in all agents are synchronized with each other.

3.4 Global execution

The semantic function for the execution of a hybrid agent nondeterministically chooses in each step either an update round or a time round, as shown by the following pseudo-code segment.

```
State macroStep (Agent a, State s) {
    [] return updateRound(a, s);
    [] return timeRound(a, s); }
```

4 Modular Simulation

The global semantics given in the previous section can be readily implemented in an algorithmic way to obtain a precise *simulation* for any hybrid system described in Charon. However, such a simulation has a big disadvantage: it is *not modular*. In other words, one can not simulate the behavior of a mode in *isolation* independent of other modes or the mode hierarchy. The lack of modularity precludes efficient implementations. For example, all flows in the previous section are synchronized on the same clock.

In this section we present an alternative, *modular simulation* for hybrid agents. This simulation may have a very efficient implementation. However, its disadvantage is that it only approximates the conceptually ideal solution.

4.1 Update Rounds

In a modular simulation, the time and the update rounds of the mode of an atomic agent are constructed in a modular way from the time and the update rounds of its submodes. The state passed along the modes is *automatically coerced* to the appropriate state for that mode, i.e., a mode can only access that part of the state that corresponds to its own variables. In programming languages terminology, the `discreteUpdate` and the `timeRound` functions are *polymorphic*.

In the modular version we do not have to work with path prefixed variables and points because the structure of a hierarchical mode is not destroyed (flattened). Moreover, in this case each mode has its own history variable, keeping a tuple: the last visited submode and its associated point. The modular version of the discrete update function is shown below. The initialization round of a mode is obtained by calling `discreteUpdate` at the initialization entry point.

```
Point×State discreteUpdate (Mode m, Point p, State s){
   Mode md = m; Point pt =p; State st = s;

   repeat {                          //loop
      if (md = m & pt = de)          //control is at default entry point
         (md, pt) = s.hs;            //execute default entry transition
      else                           //control is at regular entry
         (md, pt, st) = forany (t in enabledFanOut(md,pt, st))
         t(md, pt, st);              //execute transition
      if (md = m & pt in exitPts(m)) //control reached exit point
         return (pt, st);            //done
      else                           //control reached submode
         (pt, st) = discreteUpdate(md, pt, st);
   until (enabledFanOut(md, pt, st) = {}); }
   s.hs = (md, pt);                  //update history
   return (dx, st); }                //done
```

4.2 Time Rounds

Taking the idea of modularity seriously, in a time round each agent should be able to integrate independently of the other agents, and the integration inside a submode should be done independently of its supermodes.

The independent integration of the subagents in a composite agent, or equivalently the integration of the top modes of the associated flattened agent, is the topic of the next section. In this section we are concerned with the hierarchical integration for a mode. The main goal is to allow the modes to integrate at different speeds without compromising too much the ideal solution.

Our main assumption is that the integration speed of the parent mode is of an order of magnitude *slower* than the integration speed of the submodes. In this case, we may assume that the values integrated in the parent mode, remain constant while the submodes perform their own integration. For example, in Figure 2 left, we assume that the integration speed for `x` is slower than the integration speed for `y` that is also slower than the integration speed for `z`. This idea is shown algorithmically below.

The time round function gets as input the mode, the state, the simplified invariants of its parent mode and the integration step of its parent mode. It first computes the current submodes and the set of invariants. Then it enters the integration loop. In this loop it first simplifies the invariants according to the variables integrated in its supermode (their values are assumed to be fixed) and if the loop was traversed at least once, according to the variables declared in this mode or above but integrated in the submodes. Then it predicts its own integration step.

```
State×Time timeRound(Mode m, State s, Invariants i, Time t){
   State st; Mode md; Time d, dt;      //declare local variables
   Invariants inv = getInv(m) ∪ i;     //get invariants
   (md, pt) = s.hs;                    //get active submode and point

   for (Time tm = 0; tm < t; tm = tm + dt) {         //while time left
      inv = simplify(s, inv);          //simplify invariants
      dt = predict(inv, s, getConstraints(m), tm);//predict dt
      (st, d) = timeRound(md, s, inv, dt);         //execute submode
      st = integrate(st, getConstraints(m), d);    //integrate
      if (d < dt | violated(inv, st, tm+d))
         return (st, tm + d); }        //violation return
   return (st, tm); }                  //normal return
```

Then it calls its current submode (known from the history variable) to execute a time round. It also constrains the integration time of the submode by passing its own simplified invariants. When the submode returns, the mode synchronizes its own differential variables with the differential variables owned by the submodes by performing the integration step. If the submode returned before the assigned integration time or the invariant of the mode was violated, the mode itself returns. Otherwise it returns normally. In this way, all variables are synchronized up to the top level.

4.3 Global Execution

In the modular simulation of the global execution we want to be able to integrate each sub-agent of a composite agent (or equivalently each mode of the corresponding flattened agent) at a possibly different speed and along intervals of different length. This however inevitably leads to an out of synchronization between the agents, because as long as an agent is integrating it cannot become aware of the changes produced by the other agents.

The main idea of our approach is to keep the out of synchronization interval between agents bounded, even if the agents proceed with different speeds. An intuitive analogy would be that of a rubber band that surrounds the agents and cannot be expanded more than a length, say `dt`.

For this purpose, each step in the global execution first picks up the modes with minimum and second minimum local time. For example, in Figure 2 we pick the modes $\mathtt{M}_2$ and $\mathtt{M}_1$. Then we compute the time round interval `inc` for the minimum mode such that its local time may exceed by at most `dt` the current local time of the second minimum mode. For example, in Figure 2, the increment is `inc`.

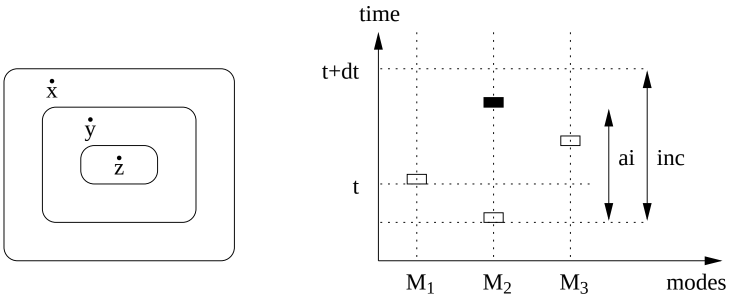

Fig. 2. Time round and global execution

The time round may end before the time interval `inc` was finished if the invariants of M_2 get violated. Hence, the time round returns, as shown in Figure 2, with an actual time increment `ai`. In this case, the mode M_2 also executes an update round to synchronize the discrete variables with the analog ones. To be able to compute the minimum and the second minimum time values and their associated modes, we keep an array of current local times of modes. This idea is presented algorithmically below.

```
Time[]×State macroStep(Time[] mTms, Agent a, State s){
   Point p; Mode[] mds = modes(a);       //initialization
   int i = getMin(mTms);                 //compute index for min.
   int j = get2ndMin(mTms);              //compute index for second min.
   m = mds[i];                           //select mode with min. time
   Time inc = mTms[j] - mTms[i] + dt;    //compute time interval

   (State s, Time ai) =
      timeRound(m, s, {}, inc);          //execute time round
   mTms[i] = mTms[i] + ai;               //update the actual time for m
   if (ai < inc) (p, s) =
      discreteUpdate(m, de, s);          //execute update round
   return (mTms, s) ; }                  //make new state and time visible
```

5 Conclusion

In this paper, we have presented a language for specification of hybrid systems that supports concurrency and hierarchy in a modular fashion. We hope that Charon is rich enough to support high-level modeling of embedded software, and is formal enough to support analysis. In this paper, we have proposed only a high-level outline for developing a modular simulator. We need to explore three orthogonal issues. First, finding a solution to a set of differential and algebraic constraints in presence of invariants requires careful detection of boundary crossings (see, for instance, [PTVF92]). Second, we handle concurrency by allowing agents to integrate separately based on their local clocks. When the guards and invariants of one agent depends on the values updated by the other agents, such a scheme may require detection and rollback. This is closely related to well understood problems concerning global states in distributed systems (see, for instance, [BT97]). Third, choosing different time scales for solving constraints at different levels of the mode hierarchy requires good heuristics to predict the step sizes. This can be done, in principle by determining the singular values of the linearized equations and scaling the equations appropriately. However, choosing a simple implicit integration scheme guarantees numerical stability and acceptable results, albeit with poor efficiencies [PTVF92].

Acknowledgments. We thank Joel Esposito and George Pappas for helpful discussions. Support from NSF grant CISE RI 9703220, NSF CARRER award CCR-9734115, DARPA/NASA grant NAG2-1214, DARPA grants ITO/MARS 130-1303-4-534328-xxxx-2000-0000, ATO/TMR DAAH04-96-1-0007, ARO grant MURI DAAH04-96-1-0007, ARO DAAG55-98-1-0393, and ARO DAAG55-98-1-0466 is gratefully acknowledged.

References

[ACH$^+$95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. 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.

[AG00] R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Proceedings of the 27th Annual ACM Symposium on Principles of Programming Languages*, 2000. To appear.

[AH97] R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In *CONCUR '97: Eighth International Conference on Concurrency Theory*, LNCS 1243, pages 74–88. Springer-Verlag, 1997.

[AHH96] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

[BJR97] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1997.

[BT97] D.P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[DGS97] A. Deshpande, A. Göllu, and L. Semenzato. Shift programming language and run-time systems for dynamic networks of hybrid automata. Technical report, University of California at Berkeley, 1997.

[GSB98] R. Grosu, T. Stauner, and M. Broy. A modular visual model for hybrid systems. In *Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'98)*, LNCS 1486. Springer-Verlag, 1998.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HHW95] T.A. Henzinger, P. Ho, and H. Wong-Toi. HyTech: the next generation. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 56–65, 1995.

[LSVW96] N. Lynch, R. Segala, F. Vaandrager, and H. Weinberg. Hybrid I/O automata. In *Hybrid Systems III: Verification and Control*, LNCS 1066, pages 496–510, 1996.

[MMP91] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-Time: Theory in Practice, REX Workshop*, LNCS 600, pages 447–484. Springer-Verlag, 1991.

[PTVF92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in FORTRAN*. Cambridge University Press, 1992.