Compositional Refinement for Hierarchical Hybrid Systems*

Rajeev Alur[1], Radu Grosu[2], Insup Lee[1], and Oleg Sokolsky[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Department of Computer Science, State University of New York at Stony Brook

Abstract. In this paper, we develop a theory of modular design and refinement of hierarchical hybrid systems. In particular, we present compositional trace-based semantics for the language CHARON that allows 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. We develop an observational trace semantics for agents as well as for modes, and define a notion of *refinement* for both, based on trace inclusion. We show this semantics to be compositional with respect to the constructs in the language.

1 Introduction

Modern software design paradigms promote *hierarchy* as one of the key constructs for structuring complex specifications. We are concerned with two distinct notions of hierarchy. In *architectural hierarchy*, a system with a collection of communicating agents is constructed by parallel composition of atomic agents, and in *behavioral hierarchy*, the behavior of an individual agent is described by hierarchical sequential composition. The former hierarchy is present in almost all concurrency formalisms, and the latter, while present in all block-structured programming languages, was introduced for state-machine-based modeling in STATECHARTS [9], and forms an integral part of modern notations such as UML [5].

A hybrid system typically consists of a collection of digital programs that interact with each other and with an analog environment. Specifications of hybrid systems integrate state-machine models of discrete behavior with differential equations for continuous behavior. This paper is about developing a *formal* and *compositional* semantics of hierarchical hybrid specifications. Formal semantics leads to definitions of *semantic* equivalence (or refinement) of specifications based on their observable behaviors, and compositionality means that semantics of a component can be constructed from the semantics of its subcomponents. Such formal compositional semantics is a cornerstone of concurrency frameworks such as CSP [11] and CCS [14], and is a prerequisite for developing modular reasoning principles such as compositional model checking and systematic design principles such as stepwise refinement.

* This research was supported in part by NSF CCR-9988409, ARO DAAG55-98-1-0466, DARPA ITO MOBIES F33615-00-C-1707, DARPA ITO MARS program, grant no. 130-1303-4-534328-xxxx-2000-0000, and ONR N00014-97-1-0505 (MURI).

The main contribution of the paper is a formal compositional semantics for the language CHARON [3] with an accompanying compositional refinement calculus. 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. 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. 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 are specified by *guarded actions* labeling transitions connecting the modes. 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.

To define the modular semantics for modes, with each mode we associate two relations, one capturing its discrete behavior and one capturing its continuous behavior. Defining the discrete relation is tricky in presence of features such as group transitions, exceptions, and history retention. Our solution relies on a closure construction, inspired by a similar construction for hierarchical discrete systems [2], which allows us to treat the transfer of control between a mode and its environment as a game.

While discrete steps of a mode and its environment are interleaved, continuous steps need to be synchronized as time is a global parameter. In fact, during a flow, all active hierarchically nested modes must participate. To allow flexible and hierarchical specifications, in CHARON, flow constraints can be specified at all levels of the hierarchy. To formalize this feature in a consistent and modular manner, we require that a mode can participate in a flow only when the control is at its default exit point. Then, all applicable constraints are properly used to define permitted flows.

The discrete and continuous relations of a mode allow us to define executions of a mode, and corresponding *traces* are obtained by projecting out the private variables. We show that the set of traces of a mode can be constructed from the traces of its submodes. This compositionality result leads to a compositional notion of refinement for modes. A mode M *refines* a mode N if they have the same interface in terms of entry/exit points and shared variables, and the traces of M is a subset of traces of N. This notion admits modular reasoning in the following manner. Suppose we obtain an implementation design I from a specification design S simply by locally replacing some submode N in S by a submode M. Then, to show I refines S, it suffices to show that M refines N. We illustrate this benefit by a simple example.

Once we have the compositionality results for modes, analogous results for agents are relatively straightforward. We define an observational trace semantics for agents, a resulting notion of refinement, and show it to be compositional with respect to the operations of parallel composition, hiding, and instantiation.

Related work. Early formal models for hybrid systems include phase transition systems [13] and hybrid automata [1]. Models such as hybrid I/O automata [12] and hybrid modules [4] allow compositional treatment of concurrent hybrid behaviors. The notion of hierarchical state machines was introduced in Statecharts [9], and is present in many software design paradigms such as Uml [5]. Our treatment of hierarchy is closest to hierarchical reactive modules [2] which shows how to define a modular semantics for hierarchical (discrete) modes. Tools such as Shift [7], Ptolemy [6], and Stateflow (see `www.mathworks.com`) allow hierarchical specifications of hybrid behavior, but formal semantics has not been a concern. HyCharts [8] presents a hierarchical model with modular operational semantics, but does not consider refinement. Masaccio [10] is a formal model for hierarchical hybrid systems. While same in spirit, it differs from our model in many technically significant aspects: it allows nesting of sequential and parallel composition, and allows a more general form of synchronous communication, but disallows high-level features of Charon modes such as exceptions, history retention, and specification of constraints at various levels.

2 Motivational example

In this section, we present a simple example that outlines features, useful in a specification language for hybrid systems. We also point out the difficulties of defining semantics for such a language. Then we give the intuition for our approach to the semantics definition, which allows us to overcome the difficulties.

Our example is a system that controls the level of liquid in a leaky tank. The level is controlled by infusing a flow of liquid into the tank. The level in the tank can be measured directly, but the rate of the leak has to be estimated. The controller has two goals: first, it must make sure that the level is within some critical bounds. If it is not, emergency measures are taken to make the level safe. When the level is safe, the controller should change the infusion rate according to instructions of the user. To do that, the controller periodically recomputes the desired rate of change for infusion and maintains the computed rate until the next update.

We now present a hierarchical description of the system in Charon. The hierarchy in Charon is twofold. The *architectural hierarchy* describes how the system agents interact with each other, hiding the details of interaction between sub-agents. The *behavioral hierarchy* describes behavior of each agent, hiding the low-level behavioral details. In our example, we have only one level of architecture description with agents `Tank` and `Controller`. There are two variables shared by the agents: `level` for the level of the liquid, and `infusion` for the infusion rate.

Both agents are *primitive*, that is, without concurrent sub-agents. Behavior of a primitive agent is given by a *mode*, a hybrid state machine equipped with analog and discrete variables. While a mode stays in a state, its analog variables are updated continuously according to a set of constraints. Taking transitions from one state to another, the mode updates its discrete variables. States of the mode are submodes that can have their own behavior. A mode has a number of *control points*, through which control enters and exits the mode. That is, to perform a computation in one of its submodes, a mode takes a transition to an entry point of that submode. When the computation is complete, a transition from an exit point of the submode is taken.

Before the computation of a mode is completed, it may be interrupted by a group transition, originating from a default exit point dx. After an interrupt, control is restored to the mode via a default entry point de. In our example, the behavior of `Tank` is represented by a single differential equation $d(level) = infusion - leak$, where *leak* is a local variable of `Tank`. Figure 1 shows the behavior of the agent `Controller`. The top-level mode of `Controller` has two submodes, `Normal` and `Emergency`. We do not show the details of the mode `Emergency`. It is activated when the level enters the critical region.

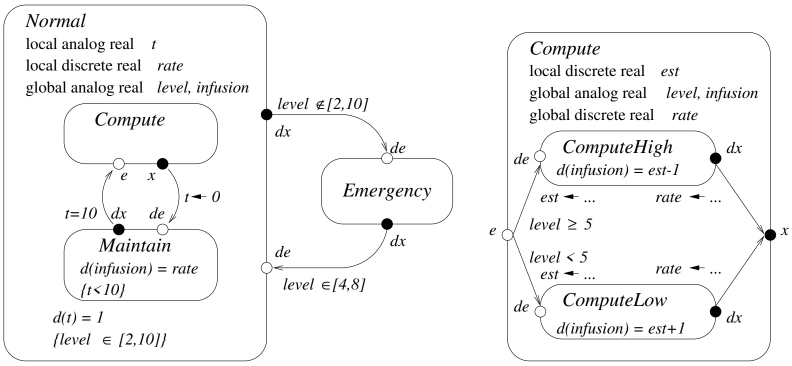

Fig. 1. Behavior of the controller

The mode `Normal` has two submodes. Submode `Maintain` is used to maintain the current rate of change for infusion, represented by a local variable *rate*. Every 10 seconds, measured by a local clock t, `Maintain` makes a call to Submode `Compute` that computes a new value of `rate`. The details of the computation are irrelevant, but we assume that the computation is done differently depending on the level. We therefore introduce two submodes in `Compute` and show only the constraints for *infusion* in each submode. The exit transition of `Compute` assigns the computed value to the variable *rate*.

Note that the mode `Normal` controls the value of the clock t, and its rate of change is the same in all its submodes. By contrast, *infusion* is updated differently in the two submodes. In this case, every submode must provide a constraint for *infusion*. Note also that *rate* is a discrete variable. It is updated only by transitions of `Compute`.

We use *invariants* to force one of the outgoing transitions. Control can reside in a mode only as long as its invariant is satisfied. As soon as an invariant is violated, control has to leave the mode by taking one of the enabled outgoing transitions. In Figure 1, invariants of the modes are shown in braces. For example, ten time units after entering the mode `Maintain` the transition to `Compute` has to be taken.

We distinguish between regular transitions and interrupts. For example, control is transferred from `Compute` to `Maintain` only when the computation is complete. When it is time to perform another computation, it will start from the beginning. On the other hand, the transition from `Normal` to `Emergency` works as an interrupt. Regardless of which submode of `Normal` is operating when an interrupt occurs, control is transferred to `Emergency`. Upon return from the interrupt, the control state of

`Normal` is restored. There is no priority between regular transitions and interrupts[1]. A mode can ignore an enabled interrupt and execute its internal transitions or let time elapse. We use invariants as described above to enforce interrupts (see the invariant of mode `Normal`). Invariants give the user finer control over interrupts. For example, a situation when an interrupt is optional for some time and then becomes urgent can be easily expressed.

In addition to discrete steps described above, a mode can make *continuous* steps, when time progresses and the analog variables of the mode are updated according to a set of constraints. Because of the hierarchical structure of the mode, the set of applicable constraints consists of the constraints defined in the mode itself and those from the currently active submode. This implies that a mode can engage in a continuous step only when its control properly resides within one of its submodes. For example, we cannot allow time to pass at the control point e of `Compute`, between executing the transition from `Maintain` to `Compute` and a transition to enter `ComputeHigh` or `ComputeLow`.

3 Modes

Notation. We will represent modes and agents as tuples of components. If T is a tuple $\langle t_1, \ldots, t_n \rangle$, we identify the component t_i of T as $T.t_i$. We extend this notation to sets of tuples. If ST is a set of tuples with the same structure, we write $ST.t_i$ to mean $\bigcup_{T \in ST} T.t_i$.

Given a set V of typed variables, a *valuation* for V is a function mapping variables to their values. We will assume that all valuations are type correct. The set of valuations over V is denoted Q_V. We will use variables s, t, possibly primed or subscripted, to range over valuations. Given a valuation s over V, and a set $W \subseteq V$, $s[W]$ denotes the restriction of s to the variables of W.

A *flow* for a set V of variables is a differentiable function f from a closed interval of non-negative reals $[0, \delta]$ to Q_V. We refer to δ as the *duration* of the flow. We assume that only constant functions are differentiable for non real-valued types. We denote a set of flows for V as $\mathcal{F}_V$.

3.1 Syntax

Definition 1. *(Mode) A* mode *M is a tuple $\langle E, X, V, SM, Cons, T \rangle$, where E is a set of* entry control points*, X is a set of* exit control points*, V is a set of variables, SM is a set of submodes, $Cons$ is a set of constraints, and T is a set of transitions.*

Variables. A mode has a finite set of typed variables V, partitioned into subsets V_a and V_d, the sets of analog and discrete variables, respectively. We also parition V into V_g and V_l, the sets of global and local variables[2]. We assume that there are no conflicts between the names of local variables of different modes.
Submodes. SM is a finite set of submodes. We require that each global variable of a submode is a variable (either global or local) of its parent mode. That is, if $N \in SM$,

[1] Other treatments of interrupts can be handled equally well within the proposed framework. For example, [2] discuss *weak* interrupts in a similar setting.
[2] Charon refines the set of global variables further according to allowed read/write access, but we won't make such a distinction in this paper for clarity of presentation.

then $N.V_g \subseteq V$. This induces a natural scoping rule for variables in a hierarchy of modes: a variable introduced as local in a mode is accessible in all its submodes but not in any other mode.

Control points. E is the set of *entry points*; X is the set of *exit points*. There are two distinguished control points representing default entry and exit: $de \in E$ and $dx \in X$. We use C for the set of all control points of the mode: $C = E \cup X \cup SM.E \cup SM.X$.

Constraints. The finite set *Cons* of constraints defines the flows permitted by M[3]. *Cons* contains an *invariant* I, which defines when the mode can be active (see the definition of an active mode below). Further, for a variable $x \in V_a$, *Cons* can contain an *algebraic* constraint A_x, which defines the set of admissible values for x, or a *differential* constraint D_x, which defines admissible values for the derivative of x with respect to time. Every invariant and an algebraic constraint is a predicate $c \subseteq Q_V$ and a differential constraint D_x is a predicate on $Q_{V \cup d(V)}$. A flow f is permitted by the mode if for every t in the domain of f, every variable in $f(t)$ satisfies all constraints in *Cons*. Examples of constraints are $d(x) \leq f(x,y)$ and $g(x,y) \leq 0$.

Transitions. T is a finite set of transitions of the form (e, α, x), where $e \in E \cup SM.X$, $x \in X \cup SM.E$, and α, the *action* of the transition, is a relation from Q_{V_g} to Q_V if $e \in E$ and from Q_V to Q_V otherwise. A transition connects control points of the mode or its submodes. When a transition is executed, it updates some variables of the mode. Every mode is assumed to have an *identity* transition from de to dx, but we disallow transitions from any non-default control point to dx. A transition that originates at a default exit point of a submode is called a *group transition* of that submode. A group transition can be executed to interrupt the execution of the submode. We require that if a submode has been exited by a group transition, it must be entered again through its default entry point to resume the interrupted execution.

Furthermore, we require that the mode cannot be blocked at any of its non-default control points. Precisely, for every e of M that is not de in M or dx in one of the submodes of M, the union α_e of all actions of the transitions originating at e is complete, that is, for every s there is t such that $(s,t) \in \alpha_e$.

Special modes. We distinguish two kinds of modes that play a special role in the semantic definitions. A mode M is a *leaf* mode if $M.SM = \emptyset$. Leaf modes perform continuous steps according to their constraints. A *top-level* mode has a single non-default entry point *init* and no non-default exit points. Top-level modes are used to describe behavior of agents, as shown in Section 4.

3.2 Semantics

Intuition. A mode can engage in a discrete or continuous behavior. During an execution, the mode and its environment either take turns making discrete steps or take a continuous step together. Discrete and continuous steps of the mode alternate. During a continuous step, the mode follows a flow from the set of flows possible for the current state for the length of its duration, updating its variables according to the flow. Note that the set of flows permitted by the mode's constraints may be further restricted by the mode's environment. A discrete step of the mode is a finite sequence of discrete steps of the submodes and enabled transitions of the mode itself. A discrete step begins in the current state of the mode and ends when it reaches an

[3] The semantics does not depend on how sets of flows are specified. Here, we chose one of the possible ways.

exit point or when the mode decides to yield control to the environment and let it make the choice of the next step. Note that in the latter case, the decision to break a discrete step is made by the mode itself. Technically, when the mode ends its discrete step in one of its submodes, it returns control to the environment via its default exit point. The closure construction, described below, ensures that the mode can yield control at appropriate moments, and that the discrete control state of the mode is restored when the environment schedules the next discrete step.

State of a mode. We define the state of a mode in terms of all variables of the mode and its submodes. We use $V_* = V \cup SM.V_*$ for the set of all variables.

The state of a mode M is a pair (c, s), where c is the location of discrete control in the mode and $s \in Q_{M.V_*}$. Whenever the mode has control, it resides in one of its control points. In this case, $c \in M.C$. We use special symbol ϵ to denote the case when the mode does not have control. Given a state (c, s) of M, we refer to c as the *control state* of M and to s as the *data state* of M.

Preemption. An execution of a mode can be preempted by a *group* transition. A group transition of a mode originates at the default exit of the mode. During any discrete step of the mode, control can be transferred to the default exit and an enabled group transition can be selected. There is no priority between the transitions of a mode and its group transitions. When an execution of a mode is preempted, the control state of the mode is recorded in a special *history* variable, a new local variable that we introduce into every mode. Then, when the mode is entered through the default entry point next time, the control state of the mode is restored according to the history variable.

The history variable and active submodes. In order to record the location of discrete control during executions, we introduce a new local variable h into each mode that has submodes. The history variable h of a mode M can assume values from the set $SM \cup \epsilon$. A submode N of M is called *active* when the history variable of M has the value N. Every top-level mode is always active.

Closure of a mode. Closure construction is a technical device to allow the mode to interrupt its execution, either to allow the environment to schedule another step or to provide for preemption of the mode execution by group transitions. Transitions of the mode are modified to update h after a transition is executed. In addition, default entry and exit transitions are added to the set of transitions of the mode. These default transitions do not affect the history variable and allow us to interrupt an execution and then resume it later from the same point.

The closure modifies the transitions of M in such a way that, after each transition, h records the active submode. If a transition leads to a control point of a submode N, the resulting state has $h = N$. Otherwise, if the transition leads to a control point of M itself, the value of h after the transition will be ϵ. For each submode N of M, the closure adds a default exit transition from $N.dx$ to $M.dx$. This transition does not change any variables of the mode and is always enabled. Default entry transitions are used to restore the local control state of M. A default entry transition leads from a default entry of the mode to the default entry of every submode N and is enabled if $h = N$. Furthermore, we make sure that the default entry transitions do not interfere with regular entry transitions originating from de. The closure changes each such transition so that it is enabled only if $h = \epsilon$.

Formally, the closure $c(M)$ of a mode $M = \langle E, X, V, SM, Cons, T\rangle$ is defined to be the mode $\langle E, X, V \cup h, c(SM), Cons, c(T)\rangle$, where $h \notin V$ is a new local variable,

$c(SM) = \{c(m) \mid m \in SM\}$ is the set of closed submodes of M, and $c(T)$ is the closed set of transitions obtained by extending T with transitions (x, α_x, dx) for every $x \in SM.dx$ and (de, α_x, e) for every $e \in SM.de$, and extending every transition in T such that

– $(s, s) \in \alpha_x$ *iff* $x \in N.E$ for some $N \in SM$ and $s[h] = N$;
– for every transition $(e, \alpha, x) \in T$, the respective closed transition is (e, α', x), where $(s, t) \in \alpha'$ *iff* $(s[V], t[V]) \in \alpha$ and
  • if $x \in N.E$ for some $N \in SM$, then $t[h] = N$, otherwise $t[h] = \epsilon$,
  • if $e \in N.X$ for some $N \in SM$, then $s[h] = N$, otherwise $s[h] = \epsilon$.

The closure construction for the example introduced in Section 2 is illustrated in Figure 2. To avoid cluttering the figure, we omit the default transitions of the submode `ComputeLow`, and do not show the variables of the modes.

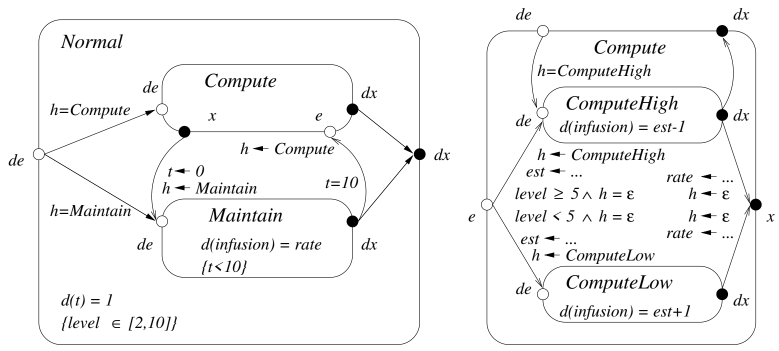

Fig. 2. Closed modes

Before formally defining executions of a mode, we illustrate continuous and discrete steps using the example in Figure 2. Assume that the the controller is in the `Maintain` mode and none of the invariants is violated. `Maintain` can voluntarily relinquish control to the environment to let it take a step or advance time by taking the default exit transition to dx of `Normal`. There, the group transition is not enabled, and the default exit transition of the parent mode is taken. When the control arrives thus at the top level, the environment can schedule a continuous step. The analog variables of all agents are updated according to the constraints of the active modes. The active modes are `Maintain`, `Normal`, and `Controller`. Thus, the applicable constraints are $d(t) = 1$ and $d(infusion) = rate$. The global variable *level* is updated according to the constraint in `Tank`. After the continuous step, control returns to `Maintain` via the chain of default entry transitions. Assume now that the invariant of `Normal` is violated while control is inside a submode of `Compute`. Then, control is transferred to dx of `Compute` and then on to dx of `Normal`. There, the choice between the group transition to `Emergency` or the default exit transition is non-deterministic. But since the invariant is violated, a continuous step cannot be taken.

Operational semantics. An operational view of a closed mode M with the set of variables V consists of a *continuous* relation R^C and, for each pair $c_1 \in E$, $c_2 \in X$, a *discrete* relation $R^D_{c_1,c_2}$.

The relation $R^C \subseteq Q_V \times \mathcal{F}_V$ gives, for every data state of the mode, the set of flows from this state. By definition, if the control state of the mode is not at dx, the set of flows for the state is empty. We require that, whenever $(s, f) \in R^C$, $f(0) = s$. In addition, for each s, the set of flows $\mathcal{F}_s = \{f \mid (s, f) \in R^C\}$ is *prefix-closed*. That is, if the domain of $f \in \mathcal{F}_s$ is $[0, \delta]$, then for every $\epsilon < \delta$, a flow $f' : [0, \epsilon]$ that coincides with f on $[0, \epsilon]$ also belongs to $\mathcal{F}_s$. R^C is obtained from the constraints of a mode and relations $SM.R^C$ of its submodes. Given a data state s of a mode M, $(s, f) \in R^C$ *iff* f is permitted by M and, if N is the active submode at s, $(s[N.V], f[N.V]) \in N.R^C$.

For each $c_1 \in E \cup SM.X$, $c_2 \in X \cup SM.E$, relation $R^D_{c_1,c_2} \subseteq Q_V \times Q_V$ describes the discrete behavior in which control is transferred from c_1 to c_2. The relation $R^D_{e,x}$ comprises *macro-steps* of a mode starting at e and ending at x. A macro step consists of a sequence of *micro-steps*. Each micro-step is either a transition of the mode or a macro-step of one of its submodes. Given the relations $R^D_{e',x'}$, $e' \in SM.E$, $x' \in SM.X$ of macro-steps of the submodes of M, a *micro-execution* of a mode $M = \langle E, X, V, SM, C, T\rangle$ is a sequence of the form $(e_0, s_0), (e_1, s_1), \ldots, (e_n, s_n)$ such that, for all i, $e_i \in C$ and $s_i \in V_*$ and for even i, $((e_i, s_i), (e_{i+1}, s_{i+1})) \in T$, while for odd i, $(s_i, s_{i+1}) \in SM.R^D_{e_i,e_{i+1}}$. Given such a micro execution of M with $e_0 = e \in E$ and $e_n = x \in X$, we have $(s_0, s_n) \in R^D_{e,x}$.

Definition 2. *(Operational semantics) The operational semantics of the mode M consists of its control points $E \cup X$, its variables V and relations R^C and $R^D_{e,x}$.*

The operational semantics of a mode defines a transition system $\mathcal{R}$ over the states of the mode. We write $(e_1, s_1) \xrightarrow{o} (e_2, s_2)$ if $(s_1, s_2) \in R^D_{e_1,e_2}$, and $(dx, s_1) \xrightarrow{f} (dx, s_2)$ if $(s_1, f) \in R^C$, f is defined on the interval $[0, t]$ and $f(t) = s_2$. We extend $\mathcal{R}$ to include *environment* steps. An environment step begins at an exit point of the mode and ends at an entry point. It represents changes to the global variables of the mode by other components while the mode is inactive. Private variables of the mode are unaffected by environment steps. Thus there is an environment step $(x, s) \xrightarrow{\varepsilon} (e, t)$ whenever $x \in X$, $e \in E$, and $s[V_p] = t[V_p]$. We let λ range over $\mathcal{F}_V \cup \{o, \varepsilon\}$. An *execution* of a mode is now a path through the graph of $\mathcal{R}$:

$$(e_0, s_0) \xrightarrow{\lambda_1} (e_1, s_1) \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_n} (e_n, s_n).$$

3.3 Trace semantics

To be able to define a refinement relation between modes, we consider a trace semantics for modes. A *trace* of the mode is a projection of its execution onto the global variables of the mode. That is, a trace is obtained from each execution by replacing every s_i with $s_i[V_g]$, and every f in transition labels with $f[V_g]$. We denote the set of traces of a mode M by L_M.

Definition 3. *(Trace semantics for modes) The trace semantics for M is given by its control points E and X, its global variables V, and its set of its traces L_M.*

In defining compositional and hierarchical semantics, one has to decide, what details of the behavior of lower-level components are observable at higher levels. In our approach, the effect of a descrete step that updates only local variables of a mode is not observable by its environment, but stoppage of time introduced by such step *is* observable. For example, consider two systems, one of which is always idle, while the other updates a local variable every second. These two systems are different, since the second one does not have flows more than one second long. Defining a modular semantics in a way that such distinction is not made seems much more difficult.

4 Agents

4.1 Syntax

Definition 4. *(Agent) An* agent $\langle TM, V, I\rangle$ *consists of a set of variables* V*, a set of initial states, and a set of top-level modes* TM*.*

The top-level modes collectively define behavior of the agent. The set V is partitioned into *local* variables V_l and *global* variables V_g. We require that $TM.V \subseteq V$, $V_g \subseteq TM.V_g$; that is, all global variables originate in some mode. The set of initial states $I \subseteq Q_V$ specifies possible initializations of the variables of the agent. A *primitive* agent has a single top-level mode. *Composite* agents have many top-level modes and are constructed by parallel composition of other agents as described below.

4.2 Semantics

An execution of an agent follows a trajectory, which starts in one of the initial states and is a sequence of flows interleaved with discrete updates to the variables of the agent. An execution of A is constructed from the relations R^C and R^D of its top-level modes. For a fixed initial state s_0, each mode $M \in TM$ starts out in the state $(init_M, s_M)$, where $init_M$ is the non-default entry point of M and $s_0[M.V] = s_M$. Note that as long as there is a mode M whose control state is at $init_M$, no continuous steps are possible. However, any discrete step of such mode will come from $R^D_{init_M, dx}$ and bring the control state of M to dx. Therefore, any execution of an agent $A = \langle TM, V, I\rangle$ with $|TM| = k$ will start with exactly k discrete initialization steps. At that point, every top-level mode of A will be at its default exit point, allowing an alternation of continuous steps from R^C and discrete steps from $R^D_{de,dx}$. The choice of a continuous step involving all modes or a discrete step in one of the modes is left to the environment. Before each discrete step, there is an environment step, which takes the control point of the chosen mode from dx to de and leaves all the private variables of all top-level modes intact. After that, a discrete step of the chosen mode happens, bringing control back to dx. Thus, an execution of A with $|TM| = k$ is a sequence $s_0 \xrightarrow{o} s_1 \xrightarrow{o} \dots s_k \xrightarrow{\lambda_1} s_{k+1} \xrightarrow{\lambda_2} \dots$ such that

- for every $0 \leq i < k$, there is $M \in TM$ such that $(s_i[M.V], s_{i+1}[M.V]) \in M.R^D_{init_M, dx}$. That is, the first k steps initialize the top-level modes of A.
- for every $i \geq k$, one of the following holds:
  - $s_i \xrightarrow{f} s_{i+1}$ such that f is defined on $[0,t]$ and $f(t) = s_{i+1}$, and for every mode $M \in TM$, $(s_i[M.V], f[M.V]) \in M.R^C$; that is, the step is a continuous step, in which every mode takes part;
  - $s_i \xrightarrow{\varepsilon} s_{i+1}$ such that for every mode $M \in TM$, $s_i[M.V_p] = s_{i+1}[M.V_p]$; that is, the step is an environment step;
  - $s_i \xrightarrow{o} s_{i+1}$ with $i > k$, there is $M \in TM$ such that $(s_i[M.V], s_{i+1}[M.V]) \in M.R^D_{de,dx}$; that is, the step is a discrete step by one of the modes.

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

Definition 5. *(Trace semantics for agents) A trace of A is an execution of A, projected onto the set of its global variables. The denotational semantics of an agent consists of its set of global variables and its set of traces.*

Let A be a primitive agent and $(init, s_0)\xrightarrow{o}(dx, s_1)\xrightarrow{\lambda_1}(c_2, s_2)\xrightarrow{\lambda_2}\ldots\xrightarrow{\lambda_{n-1}}(c_n, s_n)$ be a trace of its top-level mode. It is easy to see that $s_0\xrightarrow{o}s_1\xrightarrow{\lambda_1}s_2\xrightarrow{\lambda_2}\ldots\xrightarrow{\lambda_{n-1}}s_n$ is a trace of A. A similar statement is true for agents with multiple top-level modes.

4.3 Operations on agents

Variable hiding. The hiding operator makes a set of agent variables private. Given an agent $A = \langle TM, V, I\rangle$, the agent $A\backslash\{V_h\} = \langle TM, V', I\rangle$ with $V_l' = V_l \cup V_h, V_g' = V_g - V_h$. A trace of A, projected onto the set of global variables of $A\backslash\{V_h\}$, is a trace of $A\backslash\{V_h\}$.
Variable renaming. Variable renaming replaces a set of variables in an agent A with another set of variables. Let $V_1 = \{x_1, \ldots, x_n\}, V_2 = \{y_1, \ldots, y_n\}$ be indexed sets of variables with $V_1 \subseteq A.V$. Then, $A[V_1 := V_2]$ is an agent with the set of global variables $(A.V_g - V_1) \cup V_2$. Semantics of the variable renaming operator is given by renaming the variables in the traces of the agent.
Parallel composition. The composition of the two agents $A_1||A_2$ is an agent $A = \langle TM, V, I\rangle$ defined as follows:$A.TM = A_1.TM \cup A_2.TM, A.V_g = A_1.V_g \cup A_2.V_g, A.V_l = A_1.V_l \cup A_2.V_l$, and if $s \in A.I$ then $s[A_1.V] \in A_1.I$ and $s[A_2.V] \in A_2.I$.

5 Compositionality results

We show that our semantics is compositional for both modes and agents. First, the set of traces of a mode can be computed from the definition of the mode itself and the semantics of its submodes. Second, the set of traces of a composite agent can be computed from the semantics of its sub-agents. For the lack of space, we omit the proofs and concentrate on intuitions for the results.

5.1 Compositionality of modes

In order to show that our trace semantics for modes is compositional, we need to be able to define the semantics of a mode only in terms of the semantics of its submodes.
Compositional Trace Construction. First, we show that every trace of a mode can be constructed using the traces of the submodes.

Theorem 1. *The set of traces of a mode* M *can be computed from the set of* traces of its submodes*, its closed transition relation* $c(\mathrm{T})$ *and the set of constraints* Cons.

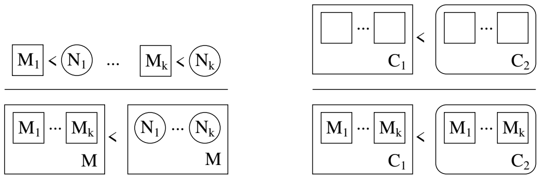

Fig. 3. Compositionality rules for modes

Theorem 1 relies on the following observation. Given a submode N of M, we can "project" a trace σ of M onto N and obtain a trace of N. This projection will 1) restrict all data states and flows to the global variables of N, 2) replace every subsequence of σ where N is inactive into a single environment step, and 3) convert continuous steps of M into continuous steps of N by removing transitions from $N.dx$ to $M.dx$ and from $M.de$ to $N.de$. The critical point in proving this observation is that, whenever the control state is at dx of M, and N is the active submode of M, N has its control state at $N.dx$, since only default exit transitions and the identity transition of the mode can end at dx.

Mode Refinement. The trace semantics leads to a natural notion of refinement between modes: a mode M refines N if it has the same global variables and control points, and every trace of M is a trace of N.

Definition 6. *(Refinement) A mode M and a mode N are said to be* compatible *if $M.V_g = N.V_g$, $M.E{=}N.E$ and $M.X{=}N.X$. Given two compatible modes M and N, M refines N, denoted $M{\preceq}N$, if $L_M{\subseteq}L_N$.*

For a finite index set I, we write $\{M_i \mid i \in I\} \preceq \{N_i \mid i \in I\}$ if $M_i \preceq N_i$ for each $i \in I$. The refinement operator is compositional with respect to the encapsulation:

Theorem 2. *(Submode compositionality) Given a mode N, suppose $SM \preceq SN$ and let $M = N[SM/SN]$. Then $M \preceq N$.*

The refinement rule is explained visually in Figure 3, left. If we consider a submode N within a mode M, the remaining submodes of M and the transitions of M can be viewed as an environment or *mode context* for N. In other words, a context for $N_1 \dots N_k$ is a mode $M[G_1, \dots G_k]$ with *holes* or *most general submodes* $G_i, 1{<}i{<}k$ that have the same interface as N_i, have no local variables and put no constraints on the update of global variables. Two contexts are said to be *compatible* if they are compatible as modes and they also are compatible on their holes.

Definition 7. *(Context traces) An* execution *of a mode context C with holes $G_1 \dots G_k$ is a path*

$$(e_0, s_0)\xrightarrow{\lambda_1}(e_1, s_1)\xrightarrow{\lambda_2}\dots\xrightarrow{\lambda_n}(e_n, s_n)$$

through the graph of $\mathcal{R}$ of C with $\lambda_i = \epsilon$ for each e_i, e_{i+1} such that e_i is in $C.X$ and e_{i+1} is in $C.E$ or e_i is in $G_j.E$ and e_{i+1} is in $G_j.X$, for $1{<}j{<}k$. A trace *of C is obtained by projecting an execution on its global variables.*

As with modes, the set of traces of a context C is denoted by L_C and *refinement* is defined by language inclusion. Given a context C with holes $G_1, \dots G_k$ and a set of modes $N_1, \dots N_k$ such that $N_i \preceq G_i$ for $1{<}i{<}k$, we write $C[N_1, \dots N_k]$ the mode obtained by filling the holes G_i of C with N_i. Contexts are also compositional.

Theorem 3. *(Context compositionality) Let C_1 and C_2 be compatible contexts with holes $G_1 \ldots G_k$. If $C_1 \preceq C_2$ then $C_1[N_1, \ldots, N_k] \preceq C_2[N_1, \ldots, N_k]$ for any set $N_i, 1<i<k$ of modes compatible with the holes, i.e., $N_i \preceq G_i$ for all i.*

A visual representation of this rule is shown in Figure 3, right. The compositionality rules allow us to decompose the proof obligation into refinement of submodes in the most general context, and refinement of contexts under the most general submode.

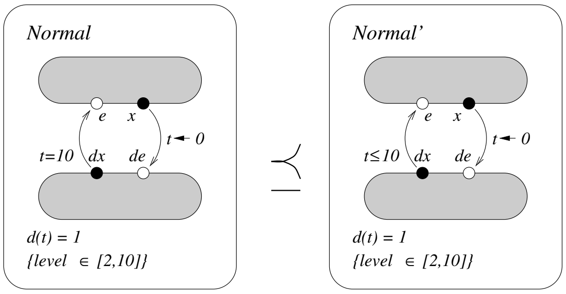

Fig. 4. Refinement example

Consider mode `Normal` in Figure 1 as a two-place context. Let `Normal'` differ from `Normal` only by allowing rate computation to happen more often. The transition to `Compute` has a relaxed guard $t \leq 10$, as shown in Figure 4. By Theorem 3, `Normal[Maintain,Compute]` $\preceq$ `Normal'[Maintain,Compute]`. If `Controller'` is the agent in which `Normal'` replaces `Normal`, then by Theorem 2, `Controller` $\preceq$ `Controller'`.

5.2 Compositionality of agents

An agent is, in essence, a set of top level modes that interleave their discrete transitions and synchronize their flows, the compositionality results for modes lift in a natural way to agents too. The operations on agents are compositional with respect to refinement.

Definition 8. *(Refinement) An agent A and an agent B are said to be* compatible *if $A.V_g = B.V_g$. Agent A refines a compatible agent B, denoted $A \preceq B$, if $L_A \subseteq L_B$.*

Theorem 4. *(Agent compositionality) Given compatible agents such that $A \preceq B, A_1 \preceq B_1$ and $A_2 \preceq B_2$. Let $V_1 = \{x_1, \ldots, x_n\}, V_2 = \{y_1, \ldots, y_n\}$ be indexed sets of variables with $V_1 \subseteq A.V$ and let $V_h \subseteq A.V$. Then $A \backslash \{V_h\} \preceq B \backslash \{V_h\}, A[V_1 := V_2] \preceq B[V_1 := V_2]$ and $A_1 || A_2 \preceq B_1 || B_2$*

In our example, `Tank`||`Controller` $\preceq$ `Tank`||`Controller'` by Theorem 4.

6 Conclusions

We have presented a hierarchical modular semantics for hybrid systems. The proposed semantics is compositional both with respect to the system architecture (parallel agents and their subagents) and the system behavior (modes and their submodes). We have introduced the notion of refinement between the system components - both modes and agents - and showed that, in the proposed semantics, composition of components preserves refinement.

We are currently working to build upon the presented compositionality results and provide assume-guarantee proof rules for hybrid systems, extending the results of [2]. The proposed semantics have been used in the modeling language Charon [3] and its toolkit, currently under development by the authors. For further details, see `http://www.cis.upenn.edu/mobies/charon/`.

References

1. 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.
2. R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Proceedings of the 27th Annual ACM Symposium on Principles of Programming Languages*, pages 390–402, 2000.
3. R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specifications of hybrid systems in CHARON. In *Hybrid Systems: Computation and Control, Third International Workshop*, volume LNCS 1790, pages 6–19, 2000.
4. 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.
5. G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1997.
6. J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Overview of the Ptolemy project. Technical Report UCB/ERL M99/37, University of California at Berkeley, 1999.
7. A. Deshpande, A. Göllu, and P. Varaiya. SHIFT: a formalism and a programming language for dynamic networks of hybrid automata. In *Hybrid Systems V*, LNCS 1567. Springer, 1996.
8. R. Grosu, T. Stauner and M. Broy. A Modular Visual Model for Hybrid Systems. In *FTRTFT'98: Formal Techniques in Real Time and Fault Tolerant Systems*, LNCS 1486. Springer-Verlag, 1998.
9. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
10. T.A. Henzinger. Masaccio: a formal model for embedded components. In *TCS 00: Theoretical Computer Science*, LNCS 1872, pages 549–563. Springer, 2000.
11. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
12. 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.
13. 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.
14. R. Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer-Verlag, 1980.