July 2006

Compositional Modeling for Refinement for Heirarchical Hybrid Systems

Rajeev Alur
University of Pennsylvania, alur@cis.upenn.edu

Radu Grosu
State University of New York at Stony Brook

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Recommended Citation
Rajeev Alur, Radu Grosu, Insup Lee, and Oleg Sokolsky, "Compositional Modeling for Refinement for Heirarchical Hybrid Systems", *Journal of Logic and Algebraic Programming* 68(1-2), 105-128. July 2006. http://dx.doi.org/10.1016/j.jlap.2005.10.004

Compositional Modeling for Refinement for Heirarchical Hybrid Systems

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.

Keywords
Hybrid Systems, Formal Methods, Compositional semantics, Model refinement

Compositional Modeling and Refinement for Hierarchical Hybrid Systems

Rajeev Alur [a] Radu Grosu [b] Insup Lee [a] Oleg Sokolsky [a,*]

[a] *Department of Computer and Information Science, University of Pennsylvania*

[b] *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.

Key words: Hybrid Systems, Formal Methods, Compositional Semantics,Model Refinement

1 Introduction

We present an approach for hybrid modeling of complex reactive systems. 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*

* This research was supported in part by NSF CCR-9988409,NSF CCR-0209024, ARO DAAD19-01-1-0473, CAREER Award NSF CCR-0133583, DARPA ITO MOBIES F33615-00-C-1707, and NSF ITR/SY 0121431

* Corresponding author.

and *compositional* semantics of 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 [Hoa85] and CCS [Mil89], and is a prerequisite for developing modular reasoning principles such as compositional model checking and systematic design principles such as stepwise refinement.

The salient aspect of the proposed approach is *hierarchy*. 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 [Har87], and forms an integral part of modern notations such as UML [BJR97]. Most of these languages, however, treat hierarchy as a *syntactic* feature of the language. A *flattening* operator transforms a hierarchical expression into one without hierarchy. The semantics, then, is given for expressions without hierarchy. Instead, we concentrate on *modular* semantics for the behavioral hierarchy, which allows us to consider each component in isolation, constructing steps of components on higher levels of the hierarchy from the steps of their immediate subcomponents. As suggested in [AH97,AG00], such modular approach is key to the development of compositional analysis techniques.

The main contribution of the paper is a formal compositional semantics for the language CHARON [AGH+00] 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 with *synchronous* interaction between agents, *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 agent is a *mode*. A mode is basically a hierarchical state machine, that is, a mode can have submodes and transitions connecting them. We allow *sharing* of modes so that the same mode definition can be instantiated in multiple contexts. Variables of a mode 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. Entry points allow us to distinguish between different activation contexts and initialize the mode differently in different circumstances. Exit points allow us to distinguish between normal and *exceptional* termination of mode

executions. To support *interrupts*, 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 of a mode from the most recent interrupt. Discrete updates are specified by *guarded actions* labeling transitions connecting the modes. Continuous updates model passage of time. During a continuous update, some of the variables in CHARON can evolve according to constraints of three distinct kinds: *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 several relations that collectively define possible steps of a mode. One captures discrete behavior of the mode and another one captures its continuous behavior. Other relations specify how the mode can be activated and deactivated. Defining the discrete relation compositionally is tricky in presence of features such as interrupts, exceptions, and history retention. Our solution relies on the entry and exit of a mode points as means of careful accounting of distinct activation/deactivation scenarios. Moreover, interrupts in mode executions and returns from interrupts are treated via special entry and exit points, providing for a uniform treatment of all scenarios.

When defining continuous steps of a mode in a consistent and modular manner, it is important to ensure that all applicable constraints are taken into consideration. To allow flexible and hierarchical specifications, in CHARON, flow constraints can be specified at all levels of the hierarchy. We ensure that all applicable constraints are properly used to define permitted flows by requiring that a mode can participate in a flow only when the discrete control has reached the bottom of the mode hierarchy. Then, the set of applicable constraints is taken from the active atomic mode and all its ancestors in the mode hierarchy.

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 globally visible 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.

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 [MMP91] and hybrid automata [ACH+95]. Models such as hybrid I/O automata [LSVW96] and hybrid modules [AH97] allow compositional treatment of concurrent hybrid behaviors. In this work, the main focus is on compositionality of sequential behaviors. Hybrid process algebras such as [Jif94,RS03,BM05,CR05] have considered compositionality in the context of much finer equivalences such as bisimulation and failures equivalences. In this work, we concentrate on trace equivalence and refinement.

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 machines [AG00], which shows how to define a modular semantics for hierarchical (discrete) modes. However, due to the presence of continuous behaviors, the treatment of discrete steps is completely different here. Unlike [AG00], every discrete step is required to begin and end at the lowest level of the hierarchy.

Tools such as SHIFT [DGV96], PTOLEMY [EJL+03], and STATEFLOW (see `www.mathworks.com`) allow hierarchical specifications of hybrid behavior, but formal semantics has not been a concern. HYCHARTS [GSB98] presents a hierarchical model with modular operational semantics, but does not consider refinement. Masaccio [Hen00] 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.

In [AGLS01], we have presented an earlier version of the CHARON language, with a purely interleaving semantics for discrete behaviors. In this paper, we have taken an alternative approach. Both discrete and continuous steps of concurrent agents are synchronous. The synchrony in discrete steps ensures that at every time instance all concurrent agents observe the same value of every shared variable. There is much work on synchronous languages such as Esterel and Lustre (see [BCE+03] for an overview). Research in synchronous languages does not consider hybrid aspects of behavior, but studies many of the same aspects of the construction of discrete steps that arise in CHARON. Composition of discrete steps in CHARON seeks to find a middle ground between the stringent acyclicity restrictions on component dependency of Lustre

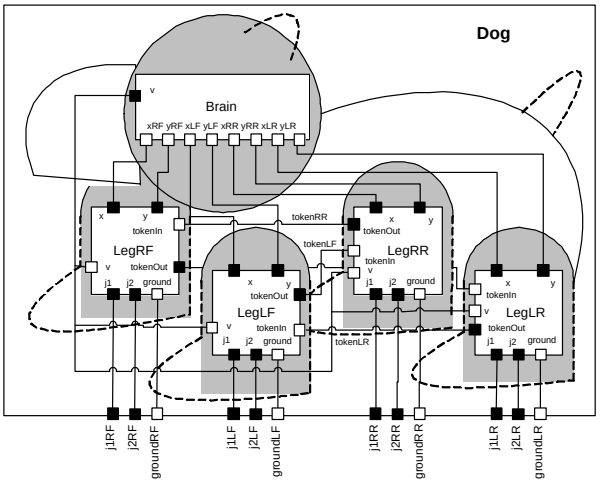

Fig. 1. Architecture of the model

and fixed-point semantics of Esterel.

2 Motivation and informal semantics

2.1 Illustrative example

We present an intuitive description of CHARON constructs and its semantics using an example taken from an on-going case study. We are designing a software controller for quadruped robots playing soccer, targeting Sony's Aibo robot dog. Below, we present a simplified controller that covers some aspects of motion planning and leg motion, but omits the components that deal with vision, inter-player coordination, etc. Walking is accomplished by moving one leg at a time, while three others remain on the ground. Legs move in the order right front, then left rear, then left front and finally right rear.

Agents and architectural hierarchy. The controller is represented as an *agent*. Agents capture architectural aspects of a model such as the composition of components and data flows. Figure 1 shows the architecture of the controller as the `Dog` agent. We distinguish between *composite* and *atomic* agents. A composite agent contains a number of sub-agents that execute concurrently and communicate by shared variables. In the example, the composite `Dog` agent contains the high-level controller `Brain` that deals with motion planning, and four low-level controllers for the leg joints. The leg agents are *instances* of the same agent `Leg`. Instances of the same agent can differ in the values of their *parameters*, and can rename variables to adjust information flows between agents.

Parameter values are specified when the agent is instantiated. Parameters indicate whether the leg is front or rear and specify joint lengths, step height, etc. Parameters are very useful for defining reusable definitions for large models. In this paper, however, we do not give semantics to parameterized specifications. Instead, we consider concrete instantiations of modes and agents, where each parameter has been replaced by a constant.

Each agent has a well-defined interface that consists of its typed input and output variables, represented visually as blank and filled squares, respectively. Connections between variables represent data flows between the agents in the model. The `Brain` agent reads variables x and y, representing leg positions, from the `Leg` agents, renaming them appropriately. That is, the variable x of agent `LegRF` is renamed to xRF in the agent `Brain`, and so on. The agent `Brain` provides the desired speed of the dog represented by the variable v, which is read by the `Leg` agents. Four boolean token variables (*tokenRF*, *tokenLF*, etc.) are shared by pairs of leg agents and are used to ensure that only one leg is in the air at any time. Each leg agent has an input variable *tokenIn* and an output variable *tokenOut*. The variable *tokenOut* of a leg agent is renamed to the same name as *tokenIn* in the next leg agent in the leg movement order. By convention, the variable *tokenOut* of the agent `LegRF` is renamed to *tokenRF*, etc. All of these variables, however, are internal to the `Dog` agent. The interface variables of the `Dog` agent are the eight output variables that represent commands sent to the joint motors in each leg, and four input variables that represent ground contact sensors in each leg.

The hierarchy of agents may be arbitrarily deep. The `Brain` agent may have several sub-agents that are concerned with separate aspects of game planning. We do not, however, show these sub-agents in detail here. An atomic agent such as `Leg` represents a single-threaded component and its behavior is given by a *mode*, described later.

Modes and behavioral hierarchy. Modes represent behavioral hierarchy in the system design. Each mode possesses a set of typed variables and describes continuous trajectories in the variable space and a single thread of discrete control. A mode can be active or inactive during an execution, depending on whether the position of discrete control is within the mode or not.

At the lowest level of the behavioral hierarchy are atomic modes. They describe purely continuous behaviors. For example, Figure 2 illustrates the behavior prescribed by the mode `UpDown`, which specifies the desired trajectory for the paw moving diagonally up or down by means of a differential constraint that asserts the relationship between the horizontal and vertical velocities of the paw, represented as the first time derivatives of the paw coordinates x and

```
mode UpDown(real dir) {
  input real v;
  output real x, y;
  diff { d(x) == 3*v; d(y) == dir*3*v; }
}
```

Fig. 2. An atomic mode

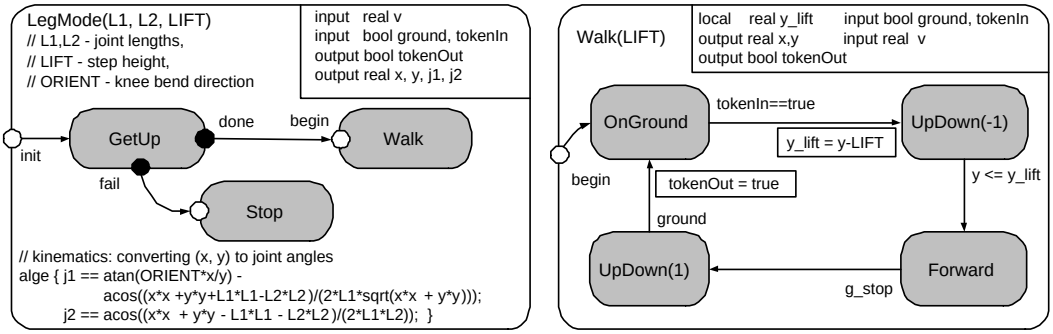

Fig. 3. Behavioral description of a leg

y, and the input variable v, representing the desired speed. Parameter *dir* is given a value when the mode is instantiated.

Additional constraints may be present on any level of the behavioral hierarchy. Besides differential constraints, modes can also have algebraic constraints and invariants. Invariants are boolean predicates that must be satisfied throughout an execution. Constraints on different levels are logically conjoined: any continuous trajectory has to satisfy the constraints of the currently active atomic mode and all of its super-modes.

Composite modes are hybrid state machines with *sub-modes* as control locations and transitions between locations that represent transfers of discrete control. Transitions have *guards* that specify when a transition can be taken, and *actions* that modify variables of the mode when the transition is taken.

Consider the mode `LegMode`, the top-level mode of the agent `Leg`, and its sub-mode `Walk`. The visual representations are shown in Figure 3. Sub-modes are shown as states labeled with the mode name. Transitions are labeled by guards and actions. To make it easier to visually distinguish between guards and actions, actions are boxed. Invariants as well as the complicated expression for the guard `g_stop`, are omitted to avoid cluttering the picture. The mode `GetUp` is entered during initialization and ensures that the dog is standing before walking begins. It has its own internal structure, which we do not discuss here. The mode `Walk` contains four sub-modes that correspond to the four segments of the leg trajectory. Note that the two sub-modes that move the leg up and down are instances of the same mode with different parameter values.

In order to specify precisely how control enters and exits a mode, we utilize the notion of *control points*. By entering a mode via different entry points, we can initialize the variables of the mode differently. Using different exit points to leave the mode, we can distinguish the normal outcome of the mode computation and different kinds of exceptions. In particular, the outcome of the `GetUp` mode may be normal, in which case it transfers control to the exit point `done`, which then proceeds to the walking mode. Alternatively, an exceptional situation may arise when the dog cannot get up by itself (for example, on an uneven surface). In that case, the execution of `GetUp` is aborted via the control point `fail` and control is transferred to the emergency mode. In either case, the computation of the `GetUp` mode is considered complete, and any re-entry of the mode starts the comutation anew. In addition to this voluntary release of control, a mode can be interrupted by a group transition, which is attached to the *default* exit point that every mode has. When an execution is interrupted, the location of discrete control is stored in the mode state so that the interrupted execution can be resumed later by entering the mode via the default entry point. In the pictorial representations, entry and exit points are denoted as blank and filled circles, respectively. Transitions incident to a default entry or exit point, which are not shown on the picture, are visually attached directly to the box representing the mode.

Note that atomic modes, such as `Forward`, have only default entry and exit points. Indeed, the only computation performed by the atomic mode is given by its differential and algebraic constraints, and the mode, by itself, does not have a notion of "completing" this computation. Any transition leaving this atomic mode interrupts the continuous behavior and is attached to the default exit point. Furthermore, since an atomic mode do not have internal control structure, no information needs to be stored when its execution is interrupted.

To ensure stability of the robot, only one leg can be in the air at any time. A leg lifts off the ground when its variable *tokenIn* get the value true. The leg then moves diagonally upwards until the desired height is reached, and the mode is switched to begin horizontal movement. When the leg is moved forward enough, another mode switch happens and the leg is moved diagonally down. When the leg reaches ground, a signal from the paw sensor sets the variable *ground*, the mode switch occurs and the token is passed to the next leg by the action of the transition.

We compare the three kinds of mode constraints using the mode `OnGround`, shown in Figure 4, as an example. A differential constraint describes the trajectory of a continuously evolving variable by specifying the value of its first derivative with respect to time. In the mode `OnGround`, the vertical position of the paw (variable y) does not change, while the horizontal position (variable

```
mode OnGround() {
  input real v;
  output real x, y;
  diff { d(x) == -1.0*v; d(y) == 0; }
  alge { tokenOut == false; }
  inv { tokenIn == false }
}
```

Fig. 4. Constraints of a mode

x), measured relative to the shoulder joint of the leg, decreases [1] . An algebraic constraint can also be used in this way, except that it specifies the value of the variable instead of its derivative. The algebraic constraints of the mode `Leg` (see Figure 3) that provide the transformation of paw position into the joint angles are used in this way. However, the algebraic constraint in mode `OnGround` is used differently. The transition of the mode `Walk` that enters the mode `OnGround` sets the value of *tokenOut* to true. The algebraic constraint, which requires *tokenOut* to be false, resets it as soon as the continuous step begins. Thus the algebraic constraint here is used as a way to implement instantaneous events. Finally, the invariant constraint allows the mode to be active only as long as the input variable *tokenIn* remains false. When it is set to true, the invariant is violated and control has to leave the mode. Here, the invariant ensures that the instantaneous event send by the preceding leg is not missed.

2.2 Informal semantics

An execution of an atomic agent, whose behavior is given by a single mode, is an alternating sequence of discrete and continuous steps. During a continuous step, the variables of the mode are updated according to the differential and algebraic constraints of the mode and its active sub-mode, recursively to the atomic mode at the bottom of the hierarchy. To ensure that all applicable constraints are used during a continuous step, we require that every discrete step of the mode begins and ends in an atomic mode. Consider the mode M shown in Figure 5. Assume that the active sub-modes are M_1 and, within it, M_{11}, and, finally, M_{111}, which is an atomic mode. In order for M to switch its active sub-mode from M_1 to M_2, it needs to traverse the following sequence of transitions. 1) leave the active atomic mode (possible when the guard g_{11} is satisfied), then 2) leave M_{11} via the exit point ex_1 (possible when the guard g_1 is satisfied), then 3) switch to M_2 if g is satisfied, and 4) enter the atomic mode M_{21}. This sequence occurs instantaneously and atomically. Alternatively, M can perform a discrete step without changing its active sub-mode. It this case, M_{11} is exited by the exit point ex_2 and the step ends in the atomic mode

[1] The body moves forward while the paw stays on the ground.

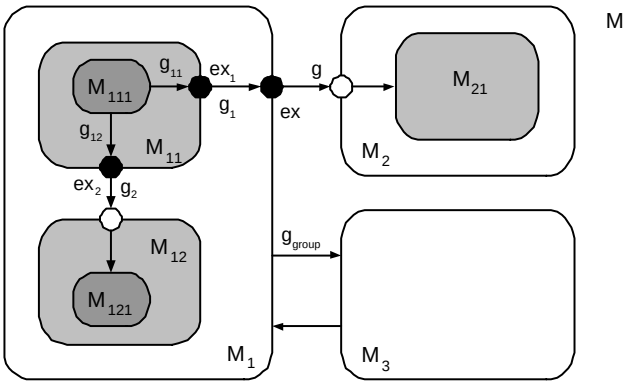

Fig. 5. Discrete steps of a mode

M_{121}. Finally, a group transition attached to the default exit point of M_1 can be taken whenever it guard g_{group} is satisfied. When that happens, control is transferred to the mode M_3 regardless of which sub-mode of M_1 was active at that moment. However, when control re-enters M_1 through the default entry point, the active sub-mode is restored.

As we will see in Section 3, steps of a mode are captured by a collection of relations. Exit relations specify how a mode can transfer control from the currently active sub-mode to a given control point. An exit step consists of an exit step of the active sub-mode followed by an exit transition leading to the specified exit point. Entry step relations specify how control is transferred from an entry point to the inside of the mode. An entry step of a mode consists of an entry transition followed by an entry step of the new active sub-mode. Finally, the internal step relation specifies the steps of a mode where the control stays within the mode. Such step is either an internal step of the active sub-mode, or an exit step of the active sub-mode, followed by a transition of the mode, followed by an entry step of the target sub-mode of the transition, which then becomes the new active sub-mode.

An execution of a composite agent A is also an alternating sequence of discrete and continuous steps. The steps are constructed from the steps of the sub-agents. In a continuous step, time progresses in all agents at the same rate and the variables are updated according to the conjunction of applicable constraints in all sub-agents. During a discrete step of A, each atomic agent that is a descendant of A in the agent hierarchy takes a discrete step. Combined together in a sequence, these discrete steps of sub-agents make a discrete step of A. When constructing discrete steps, we want to ensure that all agents participating in the step have a coherent view of the world; that is, if two agents read the value of an input variable in the same discrete step, they both get the same value. Since the discrete step of an agent may change the values of variables, we have to impose restrictions on the order of steps of sub-agents in a step of the agent. Consider the example in Figure 6. The agent A contains three sub-agents, A_1, A_2, and A_3. The agent A_1 outputs the value of the

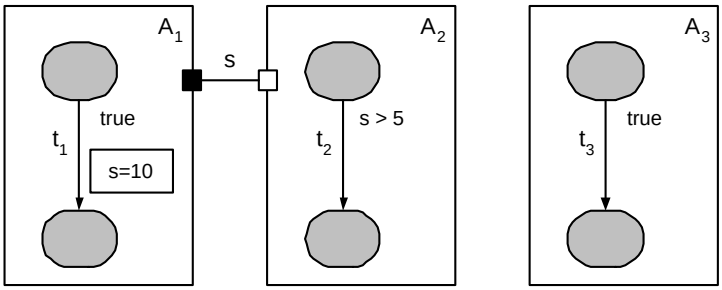

Fig. 6. Discrete steps of an agent

variable s, which is read by the agent A_2 but not by the agent A_3. Therefore, the step of A_1 has to precede the step of A_2 in any discrete step of A, and thus, the legal steps of A are $\langle t_1, t_2, t_3\rangle, \langle t_1, t_3, t_2\rangle$, and $\langle t_3, t_1, t_2\rangle$. Note that any execution ordering will produce the same result, since t_3 is independent of any variables manipulated by t_1 and t_2 and vice versa.

In order to determine the legal orderings of steps of sub-agents in a discrete step, we have to keep dependencies between variables in different agents and disallow circular dependencies to ensure that the the values assigned to variables in a discrete step are unambiguous[2]. We allow the dependencies to be dynamic in the following sense. Consider the model in Figure 1. It contains four instances of the `Leg` agent. Each instance updates the variable *tokenOut* when its leg touches the ground, which is read as *tokenIn* by the next leg in the movement order before lifting up (see Figure 3). Thus it may seem that there is a circular dependency between the legs. However, since only one leg can be in the air at any time, at most one variable is updated at any time instance. Therefore, the step of `Walk` are unambiguously constructed by making the step of the agent that corresponds to the leg in the air precede the steps of all other leg agents.

3 Syntax and semantics of modes

Notation. We will represent modes and agents as tuples of components. If T is a tuple or a set containing elements $t_1, \ldots, t_n$, we identify the component t_i of T as t_i^T. When T is understood from the context, it may be omitted.

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. Given a valuation q over V, and

[2] This is a sufficient condition. The proposed solution is a trade-off between the number of models that are syntactically rejected and the ease of implementing semantics.

a set $W \subseteq V$, $q[W]$ denotes the restriction of q to the variables of W. The value of variable v in the valuation q is denoted $q(v)$.

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

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 sub-modes, $Cons$ is a set of constraints, and T is a set of transitions.

Variables. A mode has a finite set of typed variables V, partitioned in two ways.

- The set of global variables V_g and the set of local variables V_l. We assume that there are no conflicts between the names of local variables in different modes.
- Global variables are further partitioned into the set of input variables V_i and the set of output variables V_o.

Sub-modes. SM is a finite set of sub-modes. We require that each global variable of a sub-mode is a variable (either global or local) of its parent mode. That is, if $N \in SM$, then $V_g^N \subseteq V$. This requirement ensures a natural scoping rule for variables in a hierarchy of modes: a variable introduced as local in a mode is accessible in all its sub-modes but not in any other mode. We assume the absence of name clashes between variables of the mode and local variables of its sub-modes.

Control points. E is a set of *entry points*; X is a set of *exit points*. There are two distinguished control points representing default entry and exit: $de \in E$ and $dx \in X$.

Special modes. We distinguish two kinds of modes that play a special role in the semantic definitions. A mode M is an *atomic* mode if $SM^M = \emptyset$, $T^M = \emptyset$, $E^M = \{de\}$, and $X_M = \{dx\}$. Atomic modes perform continuous steps according to their constraints and have no "interesting" discrete behaviors. 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 described in Section 4.

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 an output or local variable $x \in V_o$, *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. The invariant and algebraic constraints are predicates on Q_V and differential constraints D_x are predicates on $Q_{V \cup d(x)}$. Syntactically, an algebraic constraint A_x is a conjunction of equalities and inequalities of the form $x \bowtie f(x_1, \ldots, x_n)$, where $\bowtie = \{<, \leq, =, \geq, >\}$. A differential constraint is constructed similarly, using $d(x)$, representing the first time derivative of x, instead of x. Examples of constraints are $d(x) \leq f(x, y)$ and $g(x, y) \leq 0$. A flow f is permitted by the mode if for every $t > 0$ in the domain of f, $f(t)$ satisfies all the constraint predicates.

Transitions. Transitions of a mode connect control points of the mode and its sub-modes. A transition can originate at an entry point of a mode, or at an exit point of a submode, and lead to an exit point of the mode or an entry point of a submode. When a transition is executed, it can update variables of the mode. T is a finite set of transitions of the form (e, α, x), where $e \in E \cup X^{SM}$, $x \in X \cup E^{SM}$, and α, the *action* of the transition (see below). Each transition is categorized into *entry transitions* ($e \in E$), *exit transitions* ($x \in X$), and *internal transitions* ($e \in X^{SM}$ and $x \in E^{SM}$). A mode is not allowed to have transitions from one of its entry point directly to an exit point. It must enter one of its sub-modes first [4].

Default entry and exit points are used to handle preemption and history retention. A transition that originates at a default exit point of a sub-mode is called a *group transition* of that sub-mode. A group transition can be taken to interrupt an execution of the sub-mode. If a sub-mode has been exited by a group transition, the currently active sub-mode and the values of local variables are retained as history information. If the next time the mode is entered

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

[4] This restriction is necessary since we require that a discrete step of a mode consists of one internal transition and ends in an atomic mode. If a transition connected entry directly to an exit, the parent mode may be required to take two transitions to reach an atomic mode.

through its default entry point, the interrupted execution resumes from the saved state, as defined precisely in the next section. We disallow exit transitions of a mode leading to its default exit point so that an execution cannot be blocked if the guard of a group transition is not satisfied.

Actions of the mode transitions. Each transition has a *guard* and an *action*. A guard is a predicate over the values of the mode variables. A transition can be taken during an execution when its guard is true. An action of the transition is a sequence of assignments to the output and local variables of the mode. Each assignment is of the form $x = f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are variables of the mode. Assignments in an action are executed atomically and instantaneously when the transition is taken during an execution of the mode. Assignments are executed sequentially, that is, action $y = f_1(x); z = f_2(y)$ is the same as $y = f_1(x); z = f_2(f_1(x))$.

For assignments that make up actions of entry transitions, we have an additional restriction that if the assignment is a part of the action of an entry transition, then the assignment can depend only on the global variables of the mode and those local variables that have been assigned by previous assignments in the same action. This restriction stems from the concept widely used in programming languages: local data state of a component is constructed when the component is activated and needs to be initialized before it can be used.

We view the combination of the transition guard and action as an *action relation* α from Q_{V_g} to Q_V if $e \in E$ and from Q_V to Q_V otherwise. The pair $(q, q') \in \alpha$ if and only if q satisfies the guard of the transition and q' is the result of performing the assignments of the transition action.

3.2 Operational semantics

State of a mode. The state of a mode is a valuation of all variables of the mode and its sub-modes, V_*, computed recursively as $V_* = V_g \uplus V_{l*}, V_{l*} = V_l \uplus V_{l*}^{SM}$. We use q, possibly primed and subscripted, to denote states of a mode. Note that the history variables of the mode and its sub-modes, introduced below, collectively capture the control state.

Active mode. During a mode execution, we need to keep track of the location of discrete control. We do this using the notion of an *active* mode. At any time during an execution, one atomic mode in each agent is active, and all of its ancestors in the mode hierarchy are active as well. The top-level mode of

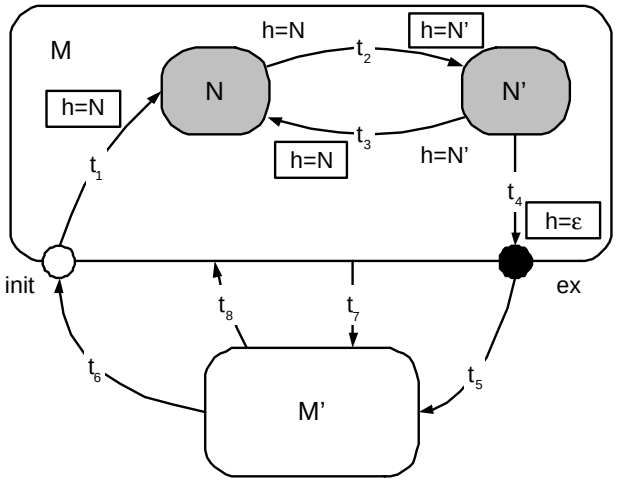

Fig. 7. Execution scenarios

each agent is always active and has one active submode. The currently active submode of a mode M is kept in a new local variable h^M that we introduce into each mode that has sub-modes. The history variable h of a mode M can assume values from the set $SM \cup \epsilon$. A sub-mode N of M is active when M is active and the history variable of M has the value N. The value ϵ represents the situation when M has completed its execution and released control to another mode. If an execution of M has been interrupted, the history variable records the submode that has been active prior to interruption and is used to continue the execution when M is reactivated.

We manipulate the history variable in the expected way by extending the action relations of each transition (e, α, x) of M. Given α, we define the new action α' as follows. Let $(q, q') \in \alpha$. If x is an entry point of a sub-mode N, then $q'(h^M) = N$, otherwise $q'(h^M) = \epsilon$. If e is an exit point of a sub-mode N', then $q(h^M) = N'$, otherwise if e is the default entry point of M, $q(h^M) \neq \epsilon$. In other words, a transition that leaves a sub-mode N can be taken only when N is active, and if the transition enters a submode N', then N' is the new active sub-mode. With entry and exit transitions, the situation is asymmetric: when an exit transition of M is taken, M becomes inactive and the history variable is ϵ. By contrast, a group transition of M interrupts the mode execution without resetting the history variable, so that the execution can be restarted when the mode is entered via the default entry point later. However, M can be exited by a group transition and then entered by a regular entry point, in which case the history information is abandoned. To allow this scenario, we do not require the history variable to be ϵ for an entry transition to occur. An interesting special case arises when the mode has been exited by a regular exit mode and thus does not have a recorded history, and is later entered via the default entry point. In this case, the mode is non-deterministically entered via one of its regular entry points.

Figure 7 illustrates the use of the history variable in different scenarios. Ignoring all other variables and transition actions, we show the value of the history variable and transitions that occur. Scenario $\{\epsilon\} \xrightarrow{t_1} \{N\} \xrightarrow{t_2} \{N'\} \xrightarrow{t_4} \{\epsilon\} \xrightarrow{t_5} \{\epsilon\} \xrightarrow{t_6} \{\epsilon\}$ is a normal execution that includes initialization and completion. Scenario $\{\epsilon\} \xrightarrow{t_1} \{N\} \xrightarrow{t_2} \{N'\} \xrightarrow{t_7} \{N'\} \xrightarrow{t_8} \{N'\} \xrightarrow{t_3} \{N\}$ represents an interrupt with a subsequent continuation of the interrupted behavior. A non-recoverable interrupt followed by reinitialization is captured by $\{\epsilon\} \xrightarrow{t_1} \{N\} \xrightarrow{t_2} \{N'\} \xrightarrow{t_7} \{N'\} \xrightarrow{t_6} \{N'\} \xrightarrow{t_1} \{N\}$. Finally, scenario $\{\epsilon\} \xrightarrow{t_1} \{N\} \xrightarrow{t_2} \{N'\} \xrightarrow{t_4} \{\epsilon\} \xrightarrow{t_5} \{\epsilon\} \xrightarrow{t_8} \{\epsilon\} \xrightarrow{t_1} \{N\}$ shows a reinitialization via a default entry: since no history information is available, the execution follows an entry transition attached to a non-deterministically chosen entry point.

Discrete steps of a mode. When viewing a mode from a higher level in the mode hierarchy, we capture three kinds of information separately: 1) we need to know what kinds of steps can the mode take without relinquishing control. Viewed externally, such a step is seen as a change in the variables of the mode. Such steps are captured by the relation R_D. Since the mode retains control, we call these steps *internal* steps of the mode. In addition, we need to know 2) how the mode can relinquish control via an exit point x, i.e., perform an *exit* step, and 3) how the mode can be acquire control via an entry point e, i.e., perform an *entry* step. Entry and exit steps are captured separately for each control point by the relations R_x and the relations R_e, respectively. All of these relations operate on the valuations of the variables of the mode.

An atomic mode has one internal step, which is the idling step enabled if and only if the invariant of the mode is satisfied. That is, for each state q such that $I(q)$, $(q, q) \in R_D$. Further, an atomic mode can be entered and exited at any time and, since it does not have entry or exit transitions, the state is not changed on entry or exit. That is, for all q, $(q, q) \in R_{de}$ and $(q, q) \in R_{dx}$.

For a composite mode M, the entry relations R_e and R_x are constructed from the actions of entry transitions and the entry relations of the sub-modes of M. For each entry transition (e, α, e'), $(q, q') \in R_e$ if, for some q'', $(q, q'') \in \alpha$ and, if e' is an entry point of a sub-mode M', $(q'', q') \in R_{e'}^{M'}$. For the default entry point, $(q, q) \in R_{de}$ whenever $q(h) \neq \epsilon$, which means that the execution of M has been previously interrupted by a group transition. When $q(h) = \epsilon$, a non-deterministic initialization occurs and thus $(q, q') \in R_{de}$ whenever $(q, q') \in R_e$ for some non-default entry point e. Similarly, for each exit transition (x', α, x) of a composite mode, $(q, q') \in R_x$ if, for some q'', $(q, q'') \in R_{x'}^{M'}$ and $(q'', q') \in \alpha$. Also, M can be interrupted by a group transition at any moment during its execution and thus has to be always ready to exit by the default exit. Therefore, for every q such that $q(h) \neq \epsilon$, $(q, q) \in R_{dx}$.

Internal steps of a composite mode M are either internal steps of the active

sub-mode or transitions of the mode that change the active sub-mode. If a transition of the mode is involved in the step, then the source sub-mode of the transition should be the active sub-mode and should allow an exit step that matches the transition, and also the target sub-mode of the transition should allow a matching entry step. Consequently, $(q, q') \in R_D$ if there exists a state q_0 that agrees with q on the values of output and local variables of $V_o \cup V_{l*}$ and

- for an active sub-mode N ($q(h^M) = N$), $(q_0[V_*^N], q'[V_*^N]) \in R_D^N$ and $q_0[V_* \backslash V_*^N] = q'[V_* \backslash V_*^N]$, or
- the following three conditions hold:
  - there exists an exit point x of the active sub-mode N such that for some q_1, $(q_0[V_*^N], q_1[V_*^N]) \in R_x^N$;
  - there exists an entry point e of a sub-mode N' such that for some q_2, $(q_2[V_*^{N'}], q'[V_*^{N'}]) \in R_e^{N'}$; and
  - there exists a transition (x, α, e) such that $(q_1, q_2) \in \alpha$.

Continuous steps. During continuous steps of a mode the control state of the mode does not change but variable values evolve continuously according to the dynamics of the mode and its active sub-mode. Continuous steps of a mode M are captured by the relation R_C. The relation $R_C \subseteq Q_V \times \mathcal{F}_V$ gives, for every state q of M, the set of flows from q. R_C is obtained from the constraints of a mode and relation R_C^N of its active sub-mode. Given a state q of a mode M, $(q, f) \in R_C$ *iff* the following three conditions hold:

- f is permitted by M,
- $(q[V_*^N], f[V_*^N]) \in N.R_C$, and
- for each variable x, $q(x) = f(0)(x)$ unless M has an algebraic constraint A_x.

Note that in the definition above, algebraic constraints can introduce discontinuities at the initial state of the flow. Otherwise, a flow from a state begins at that state.

Operational semantics. The operational semantics of the mode M consists of its control points $E \cup X$, its variables V_*, and relations R_C, R_D, R_e ($e \in E$), and R_x ($x \in X$).

3.3 Executability

The rules in the previous section for constructing steps of a mode need to be augmented with several restrictions to ensure executability of a mode. On the

one hand, constraints of a mode should always yield a non-empty set of flows. On the other hand the mode must always be able to complete a step from one atomic mode to another without being "stuck" in between.

To introduce the restrictions, we first consider the dependencies between variables in a mode. These dependencies will also be used in Section 4 to define discrete steps of an agent.

Variable dependencies in modes. We say that an output variable x is *discretely updated* in the mode M if it is assigned a new value in an action of a mode transition (for some transition t, $\exists(q, q') \in \alpha^t$, such that $q(x) \neq q'(x)$). A variable x is *continuously updated* in M if M contains an algebraic or a differential constraint for x. A variable is *accessed* in the mode M if it appears on the right-hand side of an equation or inequality within a differential or algebraic constraint, in an invariant, in a transition guard, or in the right-hand side of an assignment within a transition action.

An input variable of M is called *delayed* if it is accessed only in differential constraints of M and its sub-modes.

A variable y *depends* on a variable x if x is accessed in an action that updates y or in a constraint for y. If x is not a delayed variable, we say that y *immediately depends* on x. A dependency is continuous if y is continuously updated and discrete otherwise.

We will use graphs of immediate dependencies to define semantics of modes. A *variable dependency graph* contains mode variables as nodes and its edges are immediate dependencies between variables. The dependency graph of an atomic mode is formed by the algebraic equations of the mode. Given a composite mode M and its sub-mode M', the dependency graph $\mathcal{D}_{M'}$ for M extends the dependency graph of M' with the dependencies from the algebraic equations of M and guards and actions of the transitions that originate at M'. Given an entry point e, the dependency graph $\mathcal{D}_e$ is given by the dependencies of guards and actions of the transitions incident to e. Dependency graphs will be used to define the semantics of agents in Section 4, where restrictions will be placed on the graphs to ensure executability. Dependency graphs can be defined in several other ways. The choice of the definition is a tradeoff between the restrictiveness of the formalism (i.e. the number of models that are rejected by the imposed restrictions) and the efficiency of the execution model, since dependency graphs may have to be manipulated dynamically. A union of the dependency graphs defined above for all sub-modes M' yields a static notion of the dependency graph, which appears to be overly restrictive. On the other hand, a more dynamic notion can be defined that, depending on the values of the mode variables, does not consider the dependencies corre-

sponding to the transitions with false guards. This more dynamic dependency graph seems to be too expensive to manipulate during an execution. We chose to use the notion of the dependency graph presented above, which seems to be a reasonable compromise.

Implicit in the definition of the mode state is the active dependency graph of the mode. Whenever $q(h^M) = N$, the active dependency graph is $\mathcal{D}_N$. If $q(h^M) = \epsilon$, the mode is inactive and has no active dependency graph.

Well-defined modes. In order to ensure that the modes are well-behaved and can be used to give semantics for agents in Section 4, we impose the following requirements.

- Modes do not contain redundant variables. This means that each input variable is used and each output variable is updated in the mode or its sub-modes.
- Variables are single-writer entities, meaning a variable can be declared as output in only one agent (see Section 4). Therefore, each output variable x must be continuously updated either by the mode or by all of its sub-modes. Otherwise, we assume that the mode contains the constraint $d(x) == 0$. Both the mode and its sub-modes can continuously update an output variable, however we assume that the conjunction of constraints for each variable along a path from the mode to each of its atomic descendants has a solution.
- Each local variable of a mode is initialized by every entry transition of the mode, and every exit transition of the mode assigns every local variable a designated "undefined" value.
- Modes do not contain algebraic loops, i.e., cycles of immediate continuous dependencies in the dependency graph of algebraic constraints collected from the mode and recursively from its active sub-mode. This is necessary to ensure that the sets of algebraic constraints have continuous solutions.
- 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 sub-modes of M, the union α_e of all actions of the transitions originating at e is complete, that is, for every q there is q' such that $(q, q') \in \alpha_e$.

Simulation of a well-defined mode. A mode M that satisfies these executability conditions can be naively simulated in the following way. 1) **Initialization.** Before starting the simulation, global variables of the modes are assigned in some arbitrary way. Then, choose an entry point e to enter the mode and an entry transition that is attached to e that has its guard satisfied. Such a transition is guaranteed to exist since the mode is well-defined. Execute the assignments of the transition action. Repeat with entry transitions of the submodes, until an atomic submode is reached. 2) **Continuous step.**

Traversing the mode hierarchy from the atomic mode up to M, collect differential and algebraic constraints from each visited mode. Numerically simulate the differential constraints for one integration step, changing the variables that are differentially updated. Compute the algebraic constraints in the order of dependencies and update the algebraic variables. 3) **Discrete step.** First, collect enabled transitions. We start from the active atomic mode, which would be level 0, and go up the mode hierarchy. At each level, we separately collect internal steps and exit steps for each non-default exit point. At level 0, there is an internal idling step, or none if the invariant is violated. At each level $i > 0$, we perform the following steps. First, all internal steps collected at level $i - 1$ are used as internal steps of level i. Then, we consider the group transition and all transitions that originate at those exit points of the active submode (submode at level $i - 1$), for which there was an exit step recorded at level $i - 1$. If the transition leads to a submode, it is added to the internal steps of the current level. If the transition leads to an exit point, it is added to the exit steps for the exit point. If the invariant of the mode at level i is violated, all internal steps collected at thus far are discarded, since control has to leave the mode. Once we reach the level of M, all enabled steps are collected. We pick one of the enabled steps, and execute the actions of the transitions involved in it, updating the variables. 4) Continuous and discrete steps alternate indefinitely. Note that in a well-defined mode there is always a way to extend the execution with the next continuous or discrete step.

As an example, consider the mode in Figure 5. Assume that the active atomic submode is M_{111} and the state is such that the guard g_{11} and the invariant of M_{11} are violated and all other guards and invariants are satisfied. Then, at level 0, there is one internal idling step. At level 1 (M_{11}), all internal steps are discarded because of the violated invariant, and there is an exit step to ex_2. At level 2 (M_1), we add an internal step from ex_2, but no exit steps (since ex_1 does not offer exit steps from level 1). Finally, at level 3, we add the group transition as an internal step, and non-deterministically choose between the two enabled internal steps.

4 Syntax and semantics of agents

4.1 Syntax

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

The top-level modes collectively define behavior of the agent. The set of top-level modes in an agent represent concurrently executing threads of control

within the agent. The set V is partitioned in the same way as the variables in modes. We require that $\bigcup_{M \in TM} V^M = V$, since any variable that is not used in one of the top-level modes is useless, and $V_g \subseteq \bigcup_{M \in TM} V_g^M$, that is, each global variable of the agent originates in a top-level mode. Additionally, the agent and its top-level modes agree on their variables. The set of initial states $I \subseteq Q_V$ specifies possible initializations of the variables of the agent. An *atomic* 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.

We require that the output variables of each top-level mode are pairwise disjoint.

4.2 Operational Semantics

State of an agent. The state of an agent is a valuation of the agent variables V.

Well-defined agents. In order to be able to describe steps of an agent, we need to ensure that all variable dependencies are acyclic in each reachable state of the agent. The notion of a well-defined agent is often defined syntactically. However, in our case this notion depends on the dependency graphs of top-level modes, which are treated dynamically. Given a composite agent, we construct a graph of dependencies between the top-level modes in the following way. First, a union of the active dependency graphs is constructed. Then, the joint dependency graph is lifted to the top-level modes. That is, a top-level mode M_1 depends on a top-level mode M_2 if a variable in M_1 immediately depends on an output variable in M_2. An agent is well defined if the graph of dependencies between the top-level modes is always acyclic. Note that the dependency graph can change when a discrete step happens, since the active dependency graphs of the top-level modes can change. However, the dependency graph depends only on the set of currently active atomic modes and not on the agent state.

Discrete steps of an agent. Discrete steps of an agent A are defined by discrete steps of a its top-level modes. Each discrete step of A contains exactly one discrete step from each of its top-level modes. The steps of modes are performed sequentially in some order that is consistent with the variable dependencies in the agent. Rather then introducing this order explicitly in the definition of the step of the agent, we take a different approach that will allow us to prove compositionality in the next section.

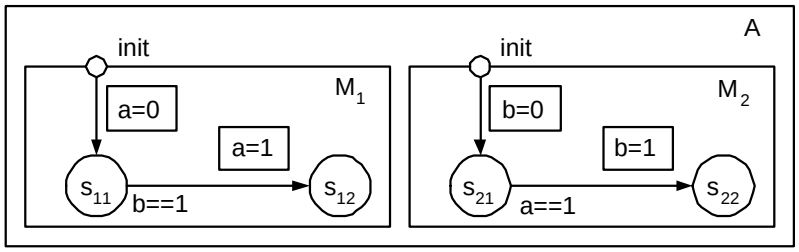

Fig. 8. A non-well-defined agent

We transform the discrete transition relation of a top-level mode into an *exposed* transition relation that reflects the fact that input variables of the mode can be modified by its environment (that is, by the steps of other top-level modes of A) before the mode takes its step. Given a top-level mode M of A with the discrete transition relation R_D^M, we define the exposed transition relation R_D^M as follows.

Let q, q' be two states of A. Then their projections on the variables of M, $q[V^M]$ and $q'[V^M]$, are the states of M. We say that M exposes a discrete transition from q to q' if M has a discrete transition to q' from a state that agrees with q on variables controlled by M (that is, its local and output variables) and agrees with q' on the input variables of M. This means that in the exposed step, M modified its variables according to its semantics, and kept the values of its input variables as they were set by the steps of other modes. Formally, we write $(q[V^M], q'[V^M]) \in \mathsf{R}_D^M$, *iff* $(q[V_*^M \backslash V_i^M] \cup q'[V_i^M], q'[V^M]) \in R_D^M$.

Now we can define discrete steps of A in terms of the exposed transition relations of its top-level modes. A has a discrete step (q, q') if for each top-level mode $M_i \in TM^A$, $(q[V^{M_i}], q'[V^{M_i}]) \in \mathsf{R}_D^{M_i}$. Initialization steps of A are constructed in a similar way using exposed initialization relations $\mathsf{R}_{init}^{M_i}$ of the top-level modes instead of their discrete relations.

Note that discrete steps of an agent are well-defined only if the agents are well-defined; that is, if variable dependency relationship is acyclic. Otherwise, the agents will be able to "guess" each other's next step. Consider, for example, the agent A in Figure 8, which is not well defined. Here, A contains two top-level modes M_1 and M_2, which control variables a and b, respectively. After performing the initialization step, we expect that A cannot engage in any discrete steps. However, $(\langle h_{M_1} = s_{11}, a = 0, b = 1\rangle, \langle h^{M_1} = s_{12}, a = 1, b = 1\rangle \in \mathsf{R}_D^{M_1}$, and $(\langle h_{M_2} = s_{21}, a = 1, b = 0\rangle, \langle h^{M_2} = s_{22}, a = 1, b = 1\rangle \in \mathsf{R}_D^{M_2}$. Therefore, A would have a counter-intuitive step, in which both agents set their variables to 1 simultaneously. Such behaviors, of course, are prevented by the acyclicity requirement on the variable dependency graph.

Continuous steps of an agent. A continuous step of an agent is a flow that is permitted by all top-level modes of the agent. That is, $q \xrightarrow{f} q'$ whenever $q[V^M] \xrightarrow{f[V^M]} q'[V^M]$ for each top-level mode $M \in TM$.

4.3 Operations on agents

To be able to construct complex agents from simpler ones, we introduce three operations on agents. The operations allow us to specify concurrent execution of agents and impose structure on communication between agents by disallowing sharing of certain variables of an agent.

Parallel composition. Parallel composition of two agents allows executions of the agents proceed concurrently, thus the top-level modes of the agents become the top-level modes of the composite agent. Variables of the two agents form the variables of the composite agent. We have to be careful to ensure that only one of the two agents can update a variable, to avoid introducing additional non-determinism into behaviors of the composite agent. Therefore, two agents are called *composable* if their output variables are disjoint. The composition of two composable agents $A_1||A_2$ is an agent $A = \langle TM, V, I\rangle$ defined as follows: $TM^A = TM^{A_1} \cup TM^{A_2}, V_g^A = V_g^{A_1} \cup V_g^{A_2}, V_l^A = V_l^{A_1} \uplus V_l^{A_2}$, and $q \in I^A$ *iff* $q[V^{A_1}] \in I^{A_1}$ and $q[V^{A_2}] \in I^{A_2}$.

Variable hiding. The hiding operator makes a set of variables local to the agent. Once a variable is local, other agents cannot read its value. This restricts unwanted communication between agents. Given an agent $A = \langle TM, V, I\rangle$ and a set of output variables $V_h \subseteq V_o$, the agent $A\backslash V_h = \langle TM, V', I\rangle$ with $V_l' = V_l \cup V_h, V_g' = V_g\backslash V_h$. A step of A, projected onto the set of global variables of $A\backslash V_h$, is a step of $A\backslash V_h$.

Variable renaming. Variable renaming changes variables names to allow agents to communicate and to avoid name clashes. When an agent is instantiated multiple times, the variables of different instances are renamed differently to avoid name clashes and to enable communication with the right agents. For example, the agent `Dog` contains four instances of the agent `Leg`. Each of the leg agents has an output variable *tokenOut*, which has to be renamed so that the legs do not interfere with each other. At the same time, input variable *tokenIn* has to be renamed in each instance to receive the token from the preceding leg. Formally, variable renaming replaces a set of variables in an agent A with another set of variables. Given an agent $A = \langle TM, V_g \cup V_l, I\rangle$, let $V_1 = \{x_1, \ldots, x_n\}, V_2 = \{y_1, \ldots, y_n\}$ be indexed sets of variables with $V_1 \subseteq V_g$

and $V_2 \cap V_l = \emptyset$. Then, $A[V_1 := V_2] = \langle TM, ((V_g \backslash V_1) \cup V_2) \cup V_l, I \rangle$. Semantics of the variable renaming operator is given by renaming the variables in the steps of the agent.

5 Compositionality results

5.1 Trace Semantics for Modes

Executions. Executions of a mode M contain steps of four kinds, given by the transition relations of M that define its operational semantics defined in Section 3.2, R_C, R_D, R_e, and R_x. An execution of M may occur within a context of a composite mode, in which M is used as a submode, or in a parallel context of an agent. When a mode is used as a sub-mode in a larger context, it may be preempted by a transition of the super-mode or relinquish control voluntarily, and then be re-entered again. Thus an execution should also capture the period of inactivity between an exit and a subsequent entry. This is accomplished by *environment steps*. The only requirement for an environment step is that it does not change the values of local variables of a mode. Thus, there is an environment step from q to q' whenever $q[V_{l*}] = q'[V_{l*}]$.

To accommodate the case of a parallel context, discrete steps have to be represented by the exposed transition relation R_D instead of R_D, as discussed in Section 4. By the same token, an exit relation R_x should be replaced by a similarly defined exposed exit relationship R_x. This is because an exit transition of M begins a discrete transition of the super-mode of M (see the construction of discrete steps in Section 3.2.

An *execution* of M of a mode, then, is a sequence

$$\ldots q_i \xrightarrow{f_i} q_{i+1} \xrightarrow{x} q_{i+2} \rightarrow q_{i+3} \xrightarrow{e} q_{i+4} \xrightarrow{f'_i} \ldots q_j \xrightarrow{f_j} q_{j+1} \xrightarrow{x'} q_{j+2} \rightarrow q_{j+3} \xrightarrow{e'} q_{j+4} \xrightarrow{f'_j} \ldots,$$

constructed as follows: after each entry step $q \xrightarrow{e} q' \in R_e$, continuous and discrete steps alternate, starting from a continuous step. An exit step $q \xrightarrow{x} q' \in \mathsf{R}_x$ may immediately follow a continuous step. Between every exit and entry step, there is exactly one environment step.

Trace semantics. 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 q_i with $q_i[V_g]$, and every f in transition labels with $f[V_g]$. We denote the set of traces of a mode M by L_M. The trace semantics

of the mode M consists of its control points $E \cup X$, its global variables V_g, and its set of traces L_M [5].

Mode contexts. The executions (and traces) of a mode M show how the mode relinquishes and acquires control from its external environment and hide this information for its submodes. To prove our compositionality results later in this section, it is necessary however, to observe how control is relinquished and acquired from its submodes, too.

Given a mode M with a submode N we call $M[N]$ a *generic mode*. The executions (and traces) of $M[N]$ externalize the interaction of M with N by inhibiting the sequential composition of M's transitions with N's entry and exit steps (and by exposing the global variables of N). Suppose we are given a discrete step $q_1 \rightarrow q_2$ in an execution of M such that $q_1(h) = N$ but $q_2(h) \neq N$. This means that the execution leaves sub-mode N. The corresponding execution of $M[N]$ will contain two steps instead: $q_1 \stackrel{x}{\rightarrow} q_1' \rightarrow q_2$ for some exit point x of N, where $(q_1, q_1') \in R_x^N$. The existence of such q_1' is guaranteed by the construction of the discrete steps of M. Similarly, discrete steps of M that enter N are also broken into two steps in $M[N]$. Hiding the internal interaction of M with N is denoted by $\overline{M[N]}$. Hence $M = \overline{M[N]}$.

Consider now a generic mode $M[N]$. The other submodes of M and the transitions of M can be intuitively viewed as a *mode context* $M[.]$ for N. To formalize this intuition, call first a mode G the *most general submode* of M compatible with N if it has: (1) the same entry and exit points as N; (2) the same global variables as N and no local variables; (3) no constraints for the entry, internal and exit relations; (4) the most general flow relation permitted by M. Then $M[.]$ is syntactic sugar for the generic mode $M[G]$.

Projection. Given a trace (or execution) σ and a mode M, the *restriction* $\sigma \Uparrow M$ is obtained from σ by: (1) keeping only the state segments that start with an entry step of M and end with an exit step of M; (2) replacing each s_i and f_i by $q_i[V^M]$ and $f_i[V^M]$; (3) each segment of σ that is not included into $\sigma \Uparrow M$ is replaced by a single environment step.

Theorem 1 (Trace construction) *Given a mode context $M[N]$. Then following holds:*

$$L_{M[N]} = \{\tau \mid \tau \in L_{M[.]} \ \wedge \ \tau \Uparrow N \in L_N\}$$

[5] We may also need to export the dynamic dependency relation.

Proof: Suppose τ is a trace of $M[N]$, and let α be a corresponding execution. Then $\alpha \Uparrow N$ is an execution of N, and hence, $\tau \Uparrow N$ is a trace of N. Moreover, τ is a trace of $M[.]$

Suppose τ is a trace of $M[.]$ and $\tau \Uparrow N$ is a trace of N. Let α be an execution of $M[.]$ corresponding to τ and β be an execution of N corresponding to $\tau \Uparrow N$. Construct now a sequence γ by replacing in α each G-subsequence with a β-subsequence (in the same order) and repeat the last value of the local variables of N in γ until the next β-subsequence. The sequence γ is then by construction an execution of $M[N]$: the G- and β-subsequences agree on the global variables and global flows, and the local variables of N are not changed by $M[.]$ or the environment. Hence, τ is a trace of $M[N]$. □

5.2 Trace Semantics for Agents

An execution of an agent is a strictly alternating sequence of continuous and discrete steps that originates in an initial state and begins with an initialization step, followed by a continuous step. A *trace* of an agent is a projection of its execution onto the global variables of A. That is, a trace is obtained from each execution by replacing every q_i with $q_i[V_g]$, and every f in transition labels with $f[V_g]$. We denote the set of traces of an agent A by L_A.

Trace semantics. The trace semantics of an agent A consists of its global variables V_g, and its set of traces L_A.

Theorem 2 (Trace construction) *Given agents A, B and sets of variables V, W. Then:*

$$
\begin{aligned}
L_{A\setminus V} &= \{\sigma[A.V \setminus V] \mid \sigma \in L_A\} \\
L_{A[V:=W]} &= \{\sigma[V := W] \mid \sigma \in L_A\} \\
L_{A\|B} &= \{\sigma[A\|B.V] \mid \sigma[A] \in L_A \ \wedge \ \sigma[B] \in L_B\}
\end{aligned}
$$

Proof: The proofs for hiding and renaming follow easily from the definition. The proof for parallel composition is as follows.

Suppose σ is a trace of $A\|B$. Then there is an execution α of $A\|B$ such that $\alpha[(A\|B).V_g] = \sigma$. Then by definition of composition, $\alpha[A]$ is an execution of A and $\alpha[B]$ is an execution of B. Hence $\sigma[A] \in L_A$ and $\sigma[B] \in L_B$.

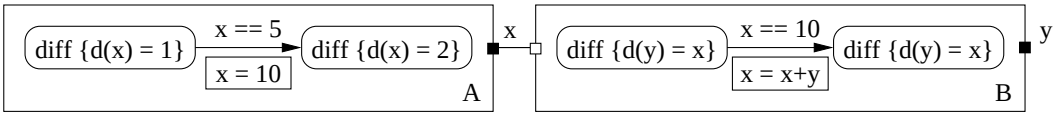

Fig. 9. Exposed relations are essential for compositionality

Suppose $\sigma[A] \in L_A$ and $\sigma[B] \in L_B$. Then there are two executions α and β of A and B such that $\alpha[A.V_g] = \sigma[A]$ and $\beta[B.V_g] = \sigma[B]$. Merge α and β into a sequence γ by taking the values of variables as updated by the agents controlling them. By construction,γ is an execution of $A\|B$ (the two agents do not update the same variables) and therefore σ is a trace of $A\|B$. □

Note. Defining the executions of top-level modes in terms of the exposed, discrete transition relation $\mathtt{R}_D$ is essential in the compositionality proof above. As an illustration, consider the agents A and B in Figure 9. A possible trace of the composed agent $A\|B$ is the following:

$$\sigma = (0,0,0) \stackrel{f,g}{\rightarrow} (5,5,12.5) \rightarrow (5,10,22.5)$$

where each state is given by a tuple $(time, x, y)$, and the flows are defined as $f = t$ and $g = t^2/2$. The trace $\sigma[B]$, which happens to be equal to σ in this case, would not be a trace of B if we were considering the relation R_D^B. This is because the tuple $((5,5,12.5),(5,10,22.5))$ is not in R_D^B. However, this tuple is by definition in the exposed relation $\mathtt{R}_D^B$.

5.3 Compositionality of Modes

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

Mode 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 $M.X = N.X$. A mode M refines a compatible mode N, denoted $M \preceq N$, if $L_M \subseteq L_N$. A context $M[N]$ is compatible with a context $P[Q]$ if M is compatible with P and N is compatible with Q. A context $M[N]$ refines a compatible context $P[Q]$, denoted $M[N] \preceq P[Q]$, if $L_{M[N]} \subseteq L_{P[Q]}$.

As shown below, refinement is compositional with respect to hiding (or mode encapsulation) and with respect to the hierarchic composition (or generic mode

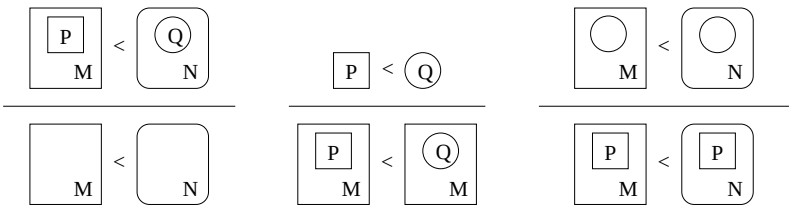

Fig. 10. Compositionality rules for modes

construction). In the latter case compositionality holds both for submodes and for mode contexts.

Theorem 3 (Compositionality of hiding) *If* $M[P] \preceq N[Q]$ *then* $\overline{M[P]} \preceq \overline{N[Q]}$.

Proof: Let $t \in L_M$. Then by definition traces, there is a trace $u \in L_{M[P]}$, such that $t = u \Uparrow M$. By hypothesis $u \in N[Q]$, too, and by compatibility of M and N, $t = u \Uparrow N$. Again by definition of traces of modes $t \in L_N$. □

Theorem 4 (Submode compositionality) *Given a mode context* $M[.]$ *for* Q *and a mode* P *such that* $P \preceq Q$. *Then* $M[P] \preceq M[Q]$.

Proof: Let $t \in M[P]$. Then by Theorem 1, $t \Uparrow P \in L_P$. By compatibility and refinement hypothesis it follows that $t \Uparrow Q \in L_Q$. Hence, by Theorem 1, $t \in P[Q]$. □

Theorem 5 (Context Compositionality) *Given a mode context* $M[.]$ *for* P *and suppose that* $M[.] \preceq N[.]$. *Then* $M[P] \preceq N[P]$.

Proof: Let $t \in L_{M[P]}$. Then by Theorem 1, $t \in L_{M[.]}$. Using now the hypothesis it follows that $t \in L_{N[.]}$. Since $t \Uparrow P \in L_P$ we conclude by Theorem 1 that $t \in L_{N[P]}$. □

The refinement rules are explained visually in Figure 10. They 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.

5.4 Compositionality of Agents

As with modes, the operations on agents are compositional with respect to refinement. Let us first define refinement for agents.

Agent 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

$L_A \subseteq L_B$.

By using Theorem 2 we show below that all agent operations are compositional with respect to refinement.

Theorem 6 (Compositionality of hiding) *If $A \preceq B$ then $A\backslash V \preceq B\backslash V$.*

Proof: Let $t \in L_{A\backslash V}$. Then by Theorem 2, there is a trace $u \in L_A$, such that $t = u[A.V\backslash V]$. By hypothesis $u \in B$, too, and therefore by Theorem 2, $t \in L_{B\backslash V}$. □

Theorem 7 (Compositionality of renaming) *If $A \preceq B$ then $A[V := W] \preceq B[V := W]$.*

Proof: Let $t \in L_{A[V:=W]}$. Then by Theorem 2, there is a trace $u \in L_A$, such that $t = u[V := W]$. By hypothesis $u \in B$, too, and therefore by Theorem 2, $t \in L_{B[V:=W]}$. □

Theorem 8 (Compositionality of composition) *If $A \preceq B$ and A is composable with C then B is composable with C and $A\|C \preceq B\|C$.*

Proof: The composability of B with C is easy to prove and therefore left out. Let $t \in L_{A\|C}$. Then by Theorem 2, $u[A] \in L_A$ and $u[C] \in L_C$. By hypothesis $u[B] \in L_B$, too, and therefore by Theorem 2, $t \in L_{B\|C}$. □

6 Conclusions

We have presented a new modular semantics for hierarchical hybrid systems. The semantics preserves data-flow dependencies between variables in the model, making it less non-deterministic than the pure interleaving approach of [AGLS01,ADE+03]. As a result, behaviors of a model are more natural from the user perspective.

The 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 on an extension of the semantics that will allow us to incorporate, in a controlled manner, the results of [AGLS01] into the framework presented here. We note that the semantic approach advocated in this paper, which preserves dependencies between updates of variables in

the model, provides a natural semantic foundation for systems in this concurrent processes are tightly coupled, such as threads running on the same processor and communicating through shared variables. For more loosely coupled systems in which processes communicate by exchanging messages, the interleaving model used in [AGLS01] appears more natural. We believe that combining the two approaches together will allow us to capture heterogeneous systems, in which both models of communication are used. By excluding certain variable dependencies from the dependency graphs, we should be able to naturally represent systems such as GALS (globally asynchronous, locally synchronous) [BCCSV03]. In doing this, we have to overcome two challenges. On the semantic level, the semantics has to remain compositional after the new features are added. This will likely lead to restrictions on how heterogeneity is manifested in the model. On the syntactic level, we have to determine how the communication model should be reflected in the system model. Our modeling approach does not represent the dependency graph directly. New syntactic classifiers for variables or data flows between variables will have to be introduced in order to capture heterogeneity. An approach to the extended semantic definition is discussed in the context of HSIF (Hybrid Systems Interchange Format) in [SLA04].

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.

[ADE+03] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 90(1):11–28, January 2003.

[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*, pages 390–402, 2000.

[AGH+00] 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 1790 of *Lecture Notes in Computer Science*, pages 6–19. Springer, 2000.

[AGLS01] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings of Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 33–48. Springer, March 2001.

[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.

[BCCSV03] A. Benveniste, L.P. Carloni, P. Caspi, and A.L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In R. Alur and Springer-Verlag I. Lee, LNCS 2855, editors, *Proceedings of the Third International Conference on Embedded Software (EMSOFT),*, volume 2855 of *Lecture Notes in Computer Science*, pages 35–50, September 2003.

[BCE+03] A. Bienveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

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

[BM05] J.A. Bergstra and C.A. Middelburg. Process algebra for hybrid systems. *Theoretical Computer Science*, 2005. To appear.

[CR05] P. J. L. Cuijpers and Michel A. Reniers. Hybrid process algebra. *Journal of Logic and Algebraic Programming*, 62(2):191–245, 2005.

[DGV96] 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.

[EJL+03] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludwig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity — the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[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, 1998.

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

[Hen00] T.A. Henzinger. Masaccio: a formal model for embedded components. In *TCS 00: Theoretical Computer Science*, LNCS 1872, pages 549–563. Springer, 2000.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Jif94] He Jifeng. From csp to hybrid systems. In *A classical mind: essays in honour of C. A. R. Hoare*, pages 171–189. Prentice Hall International (UK) Ltd., 1994.

[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.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[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.

[RS03] William C. Rounds and Hosung Song. The phi-calculus: A language for distributed control of reconfigurable embedded systems. In *Hybrid Systems: Computation and Control (HSCC'03)*, volume 2623 of *Lecture Notes in Computer Science*, pages 435–449, 2003.

[SLA04] O. Sokolsky, I. Lee, and R. Alur. Hsif semantics. Technical Report MS-CIS-04-05, Department of Computer and Information Science, University of Pennsylvania, 2004.