Hierarchical Hybrid Modeling of Embedded Systems *

R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky

University of Pennsylvania
http://www.seas.upenn.edu/hybrid/

Abstract. This paper describes the modeling language CHARON for modular design of interacting hybrid systems. The language allows specification of architectural as well as behavioral hierarchy, and discrete as well as continuous activities. The modular structure of the language is not merely syntactic, but is exploited by analysis tools, and is supported by a formal semantics with an accompanying compositional theory of refinement. We illustrate the benefits of CHARON in design of embedded control software using examples from automated highways concerning vehicle coordination.

1 Introduction

An embedded system typically consists of a collection of digital programs that interact with each other and with an analog environment. Examples of embedded systems include manufacturing controllers, automotive controllers, engine controllers, avionic systems, medical devices, micro-electromechanical systems, and robots. As computing tasks performed by embedded devices become more sophisticated, the need for a sound discipline for writing embedded software becomes more apparent (c.f [23]). Model-based design paradigm, with its promise for greater design automation and formal guarantees of reliability, is particularly attractive given the following trends.

Software Design Notations. Modern object-oriented design paradigms such as *Unified Modeling Language* (UML) allow specification of the architecture and control at high levels of abstraction in a modular fashion, and bear great promise as a solution to managing the complexity at all stages of the software design cycle [7]. There are emerging tools such as RationalRose (see `www.rational.com`) that support modeling, simulation, and code generation, and are increasingly becoming popular in domains such as automotive software and avionics.

Control Engineering. Traditionally control engineers have used tools for continuous differential equations such as MATLAB (see `www.mathworks.com`) for modeling of the plant behavior, for deriving and optimizing control laws, and for validating functionality and performance of the model through analysis and

* Supported by DARPA MoBIES grant F33615-00-C-1707

simulation. Tools such as SIMULINK recently augmented the continuous modeling with state-machine-based modeling of discrete control.

Formal Verification Tools. Model checking is emerging as an effective technique for debugging of high-level models (see [10] for a survey). Model checkers such as SMV [26] and SPIN [20] have been successful in revealing subtle errors in cache coherency protocols in multiprocessors and communication protocols in computer networks. In recent years, the model checking paradigm has been successfully extended to models with continuous variables leading to tools such as UPPAAL [22], HYTECH [18], and CheckMate [8].

This paper describes our modeling language, CHARON, that is suitable for high-level specification of interacting embedded systems. We proceed to discuss the three distinguishing aspects of CHARON.

Hybrid Modeling. Traditionally, control theory and related engineering disciplines, have addressed the problem of designing robust control laws to ensure optimal performance of processes with continuous dynamics. This approach to system design largely ignores the problem of implementing control laws as a piece of software and issues related to concurrency and communication. Computer science and software engineering, on the other hand, have an entirely discrete view of the world, which abstracts from the physical characteristics of the environment to which the software is reacting to, and is typically unable to guarantee safety and/or performance of the embedded device as a whole. An embedded system consisting of sensors, actuators, plant, and control software is best viewed as a *hybrid* system. The relevance of hybrid modeling has been demonstrated in various applications such as coordinating robot systems [2], automobiles [6], aircrafts [29], and chemical process control systems [13].

Early formal models for hybrid systems include phase transition systems [25] and hybrid automata [1]. While modularity in hybrid specifications has been addressed in languages such as hybrid I/O automata [24], CHARON allows richer specifications. Discrete updates in CHARON 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.

Hierarchical Modeling. 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 [17].

In CHARON, 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.

Compositional Semantics. Formal semantics leads to definitions of *semantic* equivalence (or refinement) of specifications based on their observable behaviors, and compositional 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 [19] and CCS [27], and is a prerequisite for developing modular reasoning principles such as compositional model checking and systematic design principles such as stepwise refinement. The global nature of time makes it challenging to define semantics of hybrid components in a modular fashion. For rich hierarchical specifications, features such as as group transitions, exceptions, and history retention, cause additional difficulties.

CHARON supports observational trace semantics for both modes and agents [4]. The key result is that the set of traces of a mode can be constructed from the traces of its submodes. This result leads to a compositional notion of refinement for modes. 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.

Overview. The remaining paper is organized as follows. In Section 2, we present the features of the language CHARON, and in Section 3 we describe the formal semantics and accompanying compositional refinement calculus. We use examples from the automotive experimental platform of the DARPA's MoBIES program for illustrative purposes. Section 4 gives a summary of the design toolkit, and we conclude in Section 5 with pointers to ongoing research on formal analysis.

2 Modeling Language

2.1 Agents and Architectural Hierarchy

We present an example from the MoBIES Automotive Open Experimental Platform (OEP) to illustrate the features of CHARON. Figures 1, 2, and 3 are CHARON agent diagrams illustrating the architectural hierarchy of a team of two vehicles.

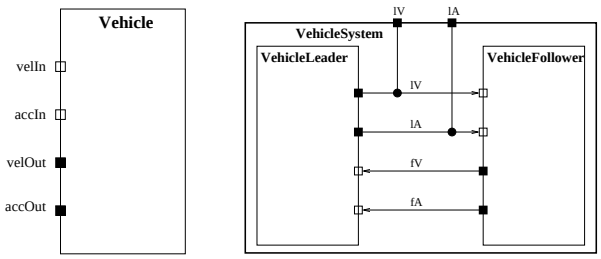

Fig. 1. The Leader-Follower Vehicle System

A single vehicle is represented by the agent `Vehicle`, shown in Figure 1(a). Each agent has a well-defined interface which consists of its typed input and output variables. In the case of `Vehicle`, `velIn` and `accIn` are the input variables, and `velOut` and `accOut` are the outputs. Formally, an *agent* consists of a set of variables V, a set of initial states, and a set of modes TM. The set V is partitioned into *local* variables V_l and *global* variables V_g; global variables are further partitioned into input and output variables. Type correct assignments of values to variables are called valuations and denoted Q_V. The set of initial states $I \subseteq Q_V$ specifies possible initializations of the variables of the agent. The modes, described in more detail below, collectively define the behavior of the agent. An *atomic* agent has a single top-level mode. *Composite* agents have many top-level modes and are constructed from other agents as described below.

Figure 1(b) illustrates the three operations defined on agents. It shows a composite agent `VehicleSystem` that contains two instances of the agent `Vehicle` *composed* in parallel. The parallel agents execute concurrently and communicate through shared variables. To enable communication between the two vehicles, global variables are *renamed*. For example, `velIn` of the follower agent and `velOut` of the leader agent are both renamed to `lV`. Finally, the communication between the vehicles can be *hidden* from the outside world. In our example, only the leader's outputs `lV` and `lA` are the outputs of the composite system. The composite agent is written in CHARON syntax as below:[1]

```
agent VehicleSystem {
 write analog real lV, lA;
 private analog real fV, fA;
 agent VehicleLeader= Vehicle[velIn,velOut,accIn,accOut := fV,lV,fA,lA];
 agent VehicleFollower= Vehicle[velIn,velOut,accIn,accOut := lV,fV,lA,fA];}
```

[1] CHARON also allows parameterized definitions. For instance, the initial position of the vehicle can be defined as a parameter within the agent `Vehicle`, and can be assigned to different values in the two instances `VehicleLeader` and `VehicleFollower`.

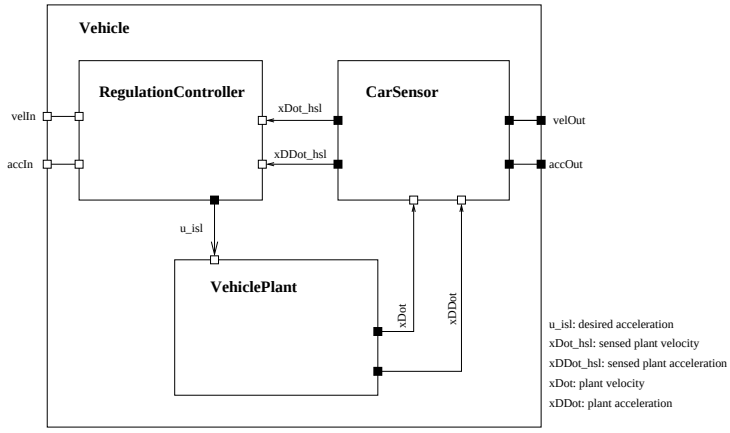

Fig. 2. The Vehicle Agent

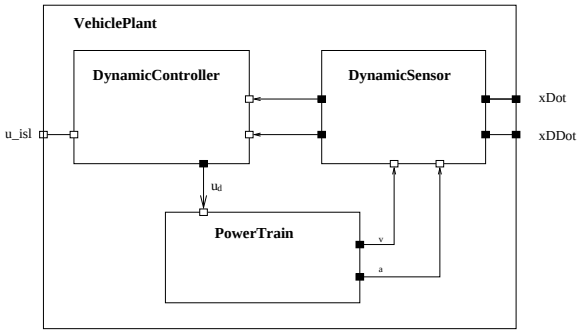

Fig. 3. The VehiclePlant Agent

The agent `Vehicle` itself has a hierarchical structure. Figure 2 illustrates the overall vehicle architecture, which comprises the regulation controller, the car sensor, and the vehicle plant. The higher level regulation controller handles data from the car sensor or other vehicles and generates a desired acceleration to the vehicle plant. The lower level dynamics controller equipped in the vehicle plant controls actual vehicle dynamics such as throttling and braking. The vehicle plant is composed of the dynamic controller, the dynamic sensor, and the powertrain. Figure 3 describes the vehicle plant. The dynamic controller maps the control command `u` onto a desired throttle position or brake command `u_d`. The sign of `u_d` will define two submodes of the powertrain: acceleration and brake. We show how the behavior of an agent is modeled in the next section.

2.2 Modes and Behavioral Hierarchy

Modes represent behavioral hierarchy in the system design. The behavior of each atomic agent is described by a mode, which corresponds to a single thread of

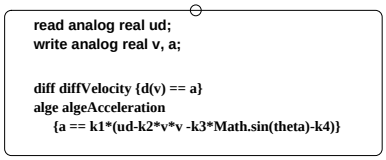

Fig. 4. The Behavior of the Agent `PowerTrain`

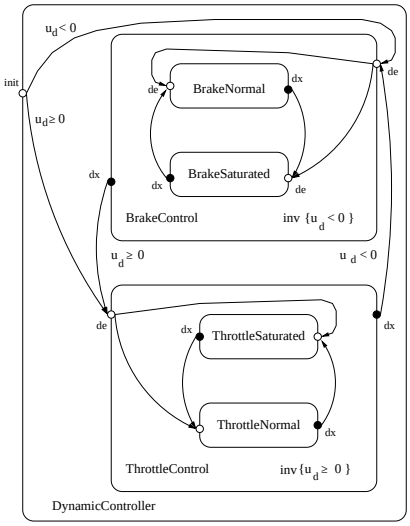

Fig. 5. The Behavior of the Agent `DynamicController`

control. At the lowest level of the behavioral hierarchy are atomic modes. They describe continuous behaviors. For example, Figure 4 illustrates the behavior of the agent `PowerTrain`. There is a differential constraint `diffVelocity` that asserts the relationship between velocity v and acceleration a: $\dot{v} = a$, and an algebraic constraint `algeAcceleration` for the acceleration, relating it to the current speed v, the control input u_d, and the road grade θ: $a = k_1 \cdot (u_d - k_2 \cdot v^2 - k_3 \cdot \sin(\theta) - k_4)$ for some constants $k_1, \ldots, k_4$.

Composite modes contain a number of submodes. During execution, a composite mode performs discrete transitions, switching between its submodes. Each (sub)mode has a well-defined data interface consisting of typed global variables used for sharing state information, and also a well-defined control interface consisting of entry and exit points that are used to connect modes modes via transitions. For example, the behavior of the agent `DynamicController` is captured by the mode shown in Figure 5. Depending on the sign of the input variable u_d that represents the acceleration desired by the higher-level controller, the dynamic controller applies either brake or throttle, represented by the submodes `BrakeControl` and `ThrottleControl`, respectively. The condition for staying in the `ThrottleControl` mode is captured by the invariant $u_d \geq 0$. When the sign

of u_d changes to negative, the invariant is violated and the controller is forced to take a transition to `BrakeControl`. Note that the mode `ThrottleControl` also has internal structure: acceleration may be normal, when a larger desired acceleration translates in more torque supplied by the engine, or saturated, when the limit of the engine capacity has been reached. The transition from `ThrottleControl` to `BrakeControl` happens regardless of which submode of `ThrottleControl` was active at that time, interrupting any lower-level behaviors.

Formally, a mode M consists of a set of submodes SM, a set of variables V, a set of *entry control points* E, a set of *exit control points* X, a set of transitions T, and a set of constraints $Cons$. As in agents, variables are partitioned into global and local variables. For the submodes of M, we require that each global variable of a submode is a variable (either global or local) of M. 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. Every mode has two distinguished control points, called default entry (de) and exit (dx) points. They are used to represent such high-level behavioral notions as interrupts and exceptions, which will be discussed in more detail in the following section.

The set $Cons$ of constraints contains constraints of three kinds. An *invariant* specifies when a mode can be active. Continuous trajectories of a variable x can be given by either an algebraic constraint A_x, which defines the set of admissible values for x in terms of values of other variables, or by a differential constraint D_x, which defines the admissible variables for the first derivative of x with respect to time.

Transitions of a mode M can be classified into *entry transitions*, which connect an entry point of M with an entry point of one of its submodes, *exit transitions*, connecting exit points of submodes to exit points of M, and internal transitions that lead from an exit point of a submode to an entry point of another submode. Every transition has a *guard*, which is a predicate over the valuations of mode variables that tells when the transition can be executed. When a transition occurs, it executes a sequence of assignments, changing values of the mode variables. 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.

In CHARON, transitions and constraints can refer to externally defined Java classes, thus allowing rich discrete and continuous specifications.

3 Formal Semantics and Compositional Refinement

We proceed to define a compositional formal semantics for CHARON. First, the operational semantics of modes and agents makes the notion of executing a CHARON model precise, and can be used, say, by a simulator. Second, we define an observational semantics for modes and agents. The observational semantics hides the details about internal structure, and retains only the information about

inputs and outputs. Informally, the observational semantics consists of the static interface (such as the global variables and entry/exit points) and dynamic interface consisting of the *traces*, that is, sequences of updates to global variables. Third, for modularity, we show that our semantics is compositional. This means that the set of traces of a component can be defined from the set of traces of its subcomponents. Intuitively, this means that the observational semantics captures *all* the information that is needed to determine how a component interacts with its environment. Finally, we define a notion of refinement (or equivalence) for modes/agents. This allows us, for instance, to relate different models of the agent `PowerTrain`. We can establish that the abstract (simplified) version of powertrain refines the detailed version, and then, to analyze the system of vehicles, use the abstract version instead of the detailed one. The compositional rules about refinement form the basis for analysis in a system with multiple components, each with a simplified and a detailed model.

3.1 Formal semantics of modes

Intuitive semantics. Before presenting the semantics formally, we give the intuition for mode executions. A mode can engage in 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 continuous trajectory that satisfies the constraints of the mode. In addition, the set of possible trajectories may be restricted by the environment of the mode. In particular, when the mode invariant is violated, the mode must terminate its continuous step and take one of its outgoing transitions. 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 exit point or when the mode decides to yield control to the environment and lets it make the choice of the next step. 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.

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 has the names of the submodes of M as values, or a special value ϵ that is used to denote that the mode does not have control. A submode N of M is called *active* when the history variable of M has the value N.

Flows. To precisely define continuous trajectories of a mode, we introduce the notion of a *flow*. 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 denote a set of flows for V as $\mathcal{F}_V$.

Syntactic restrictions on modes. In order to ensure that the semantics of a mode is well-defined, we impose several restrictions on mode structure. First, we assume that the set of differential and algebraic constraints in a mode always has a non-empty set of flows that satisfy them. This is needed to ensure that the set of behaviors of a mode is non-empty. Furthermore, we require that the mode cannot be blocked at any of its non-default control points. This means that the disjunction of all guards originating from a control point evaluates to `true`.

State of a mode. We define the state of a mode in terms of all variables of the mode and its submodes, including the local variables on all levels. We use V_* for the set of all variables. The set of local variables of a mode together with the local variables of the submodes are called the private variables and is denoted as V_p.

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, that is, $c \in M.C$. 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.

Closure of a mode. Closure construction is a technical device to allow the mode to interrupt its execution and to maintain its history variable. Transitions of the mode are modified to update the history variable h after a transition is executed. Each entry or internal transition assigns the name of the destination mode to h, and exit transitions assign ϵ to h. 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 default entry and exit transitions are added in the following way. 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 that leads from a default entry of M to the default entry of a submode N 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 $h = \epsilon$. The closure construction for the mode `DynamicController` introduced in Section 2 is illustrated in Figure 6.

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 $R^D_{c_1,c_2}$.

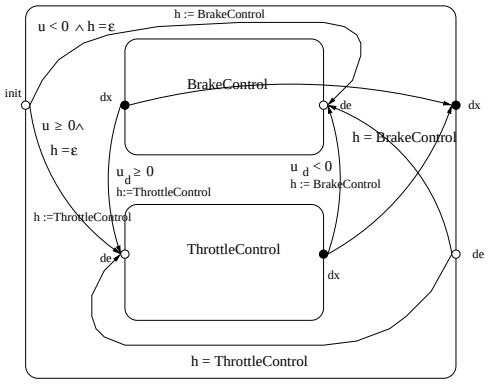

Fig. 6. Closed modes

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. 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 satisfies the constraints of M and, if N is the active submode at s, (s, f), restricted to the global variables of N, belongs to $N.R^C$.

The relation $R^D_{e,x}$, for each entry point e and exit point x of a mode, 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 $R^D_{e,x}$ of the submodes of M, a *micro-execution* of a mode M is a sequence of the form $(e_0, s_0), (e_1, s_1), \ldots, (e_n, s_n)$ such that every (e_i, s_i) is a state of M and for even i, $((e_i, s_i), (e_{i+1}, s_{i+1}))$ is a transition of M, while for odd i, (s_i, s_{i+1}) is a macro-step of one of the submodes of M. Given such a micro execution of M with $e_0 = e \in E$ and $e_n = x \in X$, we have $(s_0, s_n) \in R^D_{e,x}$. To illustrate the notion of macro-steps, consider the closed mode `DynamicController` from Figure 6. Considering only the control points, we have the following micro-execution when $u > 0$ and $u_c < maxThrottle$: *init*, *ThrottleNormal.de*, *ThrottleNormal.dx*, *dx*. For every u, u_c satisfying the above inequalities this micro-execution gives us a macro-step in $R^D_{init,dx}$.

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 $(s_1, f) \in R^C$, where 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 $\mathcal{R}$:

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

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 executions onto the global variables of the mode. The *trace semantics* for M is given by its control points E and X, its global variables V_g, 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 discrete step that updates only local variables of a mode is not observable by its environment, but stoppage of time introduced by such a 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.

3.2 Trace semantics for agents

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 mode. 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 a 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 $s_0 \xrightarrow{o} s_1 \xrightarrow{o} \ldots s_k \xrightarrow{\lambda_1} s_{k+1} \xrightarrow{\lambda_2} \ldots$ such that

- The first k steps are discrete and initialize the top-level modes of A.
- for every $i \geq k$, one of the following holds:
  - the i^{th} step is a continuous step, in which every mode takes part, or
  - the i^{th} step is a discrete environment step, or
  - the i^{th} step is a discrete step by one of the modes and the private variables of all other modes are unchanged.

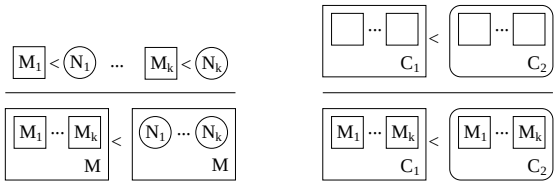

Fig. 7. Compositionality rules for modes

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.

A trace of an agent 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 V_g and its set of traces L_A.

Trace semantics for modes and agents can be related to each other in an obvious way. Given an atomic agent A whose behavior is given by a mode M, we can obtain a trace of A by taking a trace of M and erasing the information about the control points from it.

3.3 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.

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. 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 $L_M{\subseteq}L_N$.

The refinement operator is compositional with respect to the encapsulation. If, for each submode N_i of M there is a mode N_i' such that $N_i \preceq N_i'$, then we have that $M \preceq M'$, where M' is obtained from M by replacing every N_i with N_i'. The refinement rule is explained visually in Figure 7, left.

A second refinement rule is defined for contexts of modes. Informally, 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.

As with modes, refinement of contexts is also defined by language inclusion and is also compositional. If a context C_1 refines another context C_2, then inserting modes $M_1, \ldots, M_k$ into the two contexts preserves the refinement property. A visual representation of this rule is shown in Figure 7, right. Precise statements of the results can be found in [4].

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. 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 $L_A \subseteq L_B$. 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 $A_1 || A_2 \preceq B_1 || B_2$

4 The CHARON Toolkit

In this section we describe the CHARON toolkit. Written in Java, the toolkit features an easy-to use graphical user interface, with support for syntax-directed text editing, a visual input language, a powerful type-checker, simulation and a plotter to display simulation traces. The CHARON GUI uses some components from the model checker JMOCHA [3], and the plotter uses a package from the modeling tool PTOLEMY [12].

The editor windows highlight the CHARON language keywords and comments. *Parsing on the fly* can be enabled or disabled. In case of an error while typing, the first erroneous token will be highlighted in red. Further, a pop up window can be enabled that tells the user what the editor expects next. Clicking one of the pop up options, the associated text is automatically inserted at the current cursor position. This allows the user not only to correct almost all syntactic errors at typing but also to learn the CHARON language.

The CHARON toolkit also includes a visual input language capability. It allows the user to draw agent and mode definitions in a hierarchical way, as shown in Figures 1 – 5. The interpreter of the visual input translates the specification into text-based CHARON source code using an intermediate XML-based representation. The visual input tool is depicted in Figure 8.

Once an edited and saved CHARON language file exists, the user can simulate the hybrid system. In this case the CHARON toolkit calls the parser and the type checker. If there are no syntactic errors, it generates a *project context* that is displayed in a separate project window that appears on the left hand side of the desktop, as shown in Figure 9.

The project window displays the internal representation of CHARON in a convenient tree format. Each node in the tree may be expanded or collapsed by clicking it. The internal representation tree consists of two nodes: `agents` and `modes`. They are initially collected from the associated CHARON file.

A CHARON specification describes how a hybrid system behaves over the course of time. CHARON's simulator provides a means to visualize a possible behavior of the system. This information can be used for debugging or simply for understanding in detail the behavior of the given hybrid system description.

The simulation methodology used in the CHARON toolkit, which is depicted in Figure 10, resembles concepts in code generation from a specification. As CHARON allows to write external Java source code the simulator needs to be an executable Java program. CHARON has a set of Java files that represent a core

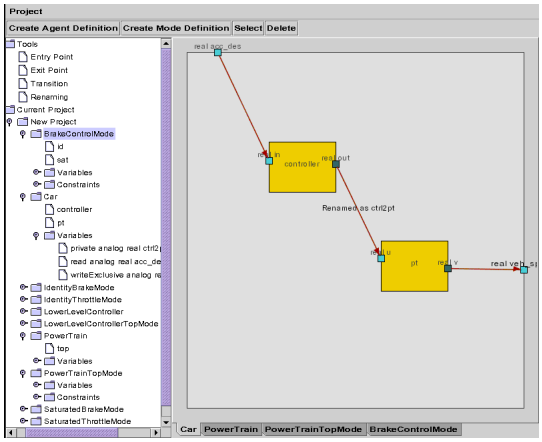

Fig. 8. The visual input tool of CHARON: The agent car is defined as the composition of two other agents. The arrows depict variable renamings.

simulator. Given a CHARON file, Java files are automatically generated which represent a Java interpretation of the CHARON specification of a hybrid system. They are used in conjunction with the predefined simulator core files and the external Java source code to produce a simulation trace.

The CHARON plotter allows the visualization of a simulation trace generated by the simulator. It draws the value of all selected variables using various colors with respect to time. It also highlights the time that selected transitions have been taken. A screen-shot of the plotter is given in Figure 11.

More information on the CHARON toolkit, along with a preliminary release, is available at `www.cis.upenn.edu/mobies/charon/`.

5 Research in Formal Analysis

Since CHARON models have a precise semantics, they can be subjected to a variety of analyses. In this final section we will give a brief overview of our ongoing research efforts in formal analysis methods for hybrid systems. These include new techniques for accurate and efficient simulation, reachability analysis to detect violations of safety requirements, and abstraction methods for enhancing the applicability of analysis techniques.

Accurate event detection. The problem of accurately detecting and localizing the occurrence of transitions when simulating hybrid systems has received an increased amount of attention in recent years. This is partially motivated by the observation that the commonly used technique of simply checking the value of the guards at integration points can cause the simulator not to detect enabling of transitions. It has been shown that such inaccuracies can lead to grossly inaccurate simulations due to the discontinuous nature of hybrid sys-

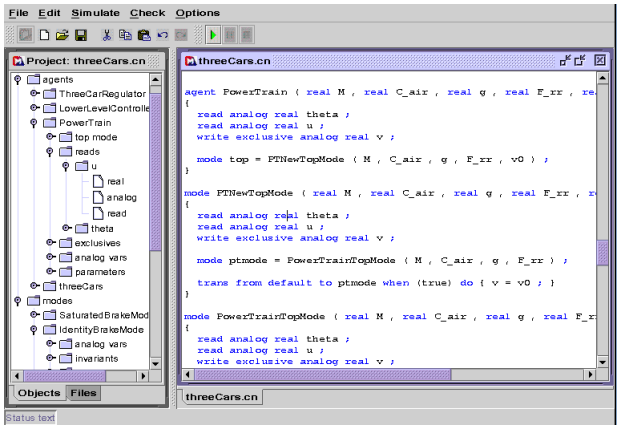

Fig. 9. The editor frame on the right hand side of the CHARON desktop and the corresponding project frame on the left

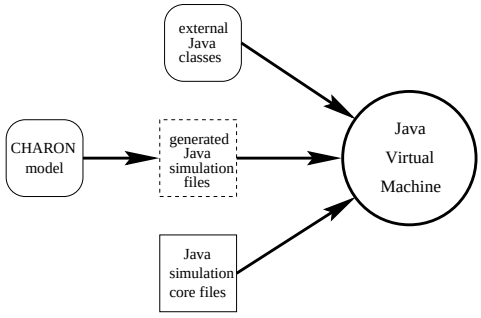

Fig. 10. The simulation methodology of CHARON

tems. We have developed a method [15] which is guaranteed to detect enabling of all transitions. Our method has the advantage of being the only method which can properly detect such events when they occur in the neighborhood of model singularities. We select our step size in such a way as to steer the system toward the event surface without overshooting it.

Multirate simulation. Many systems, especially hierarchical ones, naturally evolve on different time scales. For example the center of mass of an automobile may be accelerating relatively slow compared to the rate at which the crank shaft angle changes; yet, the evolution of the two are intimately coupled. Despite this disparity, traditional numerical integration methods force all coupled differential equations to be integrated using the same step size. The idea behind multi-rate integration method is to use larger step sizes for the slow changing sets of differential equations and smaller step sizes for the differential

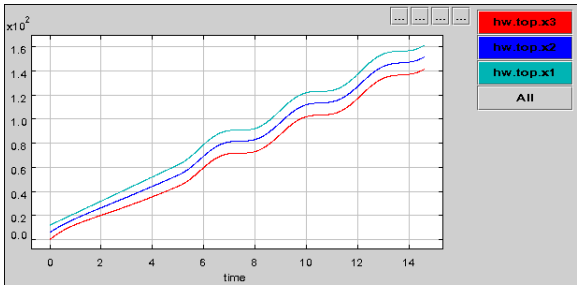

Fig. 11. A plot of a simulation trace of a three car vehicle-to-vehicle coordination model. The three graphs represent the x-coordinates of the three respective cars.

equations evolving on the fast time scale. Such a strategy increases efficiency without compromising accuracy. To implement such a schme we need to show how to accommodate coupling between the sets of fast and slow equations when they are integrated asynchronously and how to schedule the order of integration. In [14] we resolve these issues and introduce a multirate algorithm well suited to hybrid system simulation.

Distributed Simulation. Another way for simulation speed-up is to utilize more computing resources in a multi-processing platform by exploiting the inherent modularity of systems described in CHARON. Each agent of CHARON has different dynamics and thus does not have to be integrated at the same speed. The challenge is to determine an effective scheme for communication of the values of the shared variables among the various agents simulated on different processing units.

Requiem for reachability analysis. Formal verification of safety requirements of hybrid systems requires the computation of reachable states of continuous systems. Requiem is a Mathematica notebook which, given a nilpotent linear differential equation and a set of initial conditions, symbolically computes the set of reachable states exactly. Given various classes of linear differential equations and semi-algebraic sets of initial conditions, the computation of reachable sets can be posed as a quantifier elimination problem in the decidable theory of the reals as an ordered field. Given a nilpotent system and a set defined by polynomial inequalities, Requiem automatically generates the quantifier elimination problem and invokes the quantifier elimination package in Mathematica 4.0. If the computation terminates, it returns a quantifier free formula describing the reachable set. More details can be found in [21]. The entire package is available for free at `www.seas.upenn.edu/hybrid/requiem.html`.

Predicate Abstraction. Abstraction is emerging as the key to formal verification as a means of reducing the size of the system to be analyzed [11]. The main obstacle towards an effective application of model checking to hybrid systems is the complexity of the reachability procedures which require expensive computations over sets of states. For analysis purposes, it is often useful to ab-

stract a system in a way that preserves the properties being analyzed while hiding the details that are of no interest [5]. We build upon notion of predicate abstraction [28] for formal analysis of hybrid systems. Using a set of boolean predicates, that are crucial with respect to the property to be verified, we construct a finite partition of the state space of the hybrid system. The states of the abstracted system correspond to truth assignments to the set of predicates. By using conservative reachability approximations we guarantee that if the property holds in the abstracted system, then it also holds in the concrete system represented by the hybrid system. Conversely, if the property does not hold in the abstract system, then it may or may not hold in the concrete system. In the latter case, the concrete system will be checked against the counter-example found during the model checking of the abstract system. If a trace corresponding to the counter-example is feasible in the concrete system we have established the property to be false. This procedure can also help discover new predicates that can be used to refine the abstraction, as it is suggested in [9] for analysis of discrete systems. The combination of this method for finding new predicates and our abstraction procedure thus provides an effective way to apply a finite state model checking approach to hybrid systems.

Multi-robot coordination. We develop a hybrid system framework and the software architecture for the deployment of multiple autonomous robots in an unstructured and unknown environment with applications ranging from scouting and reconnaissance, to search and rescue and manipulation tasks. Our software framework allows a modular and hierarchical approach to programming deliberative and reactive behaviors in autonomous operation. Formal definitions for sequential composition, hierarchical composition, and parallel composition allow the bottom-up development of complex software systems. We demonstrate the algorithms and software on an experimental testbed that involves a group of car-like robots using a single omni-directional camera as a sensor without explicit use of odometry. More information can be found in [2, 16].

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, A. Das, J. Esposito, R. Fierro, Y. Hur, G. Grudic, V. Kumar, I. Lee, J. P. Ostrowski, G. Pappas, J. Southall, J. Spletzer, and C. J. Taylor. A framework and architecture for multirobot coordination. In *Proc. ISER00, Seventh Intl. Symp. on Experimental Robotics*, pages 289–299, 2000.
3. R. Alur, L. de Alfaro, R. Grosu, T.A. Henzinger, M. Kang, R. Majumdar, F. Mang, C.M. Kirsch, and B.Y. Wang. MOCHA: A model checking tool that exploits design structure. In *Proc. 23rd Intl. Conf. on Software Engineering*, pages 835–836, 2001.
4. R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Hybrid Systems : Computation and Control*, LNCS 2034, pages 33–48, 2001.
5. R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, July 2000.

6. A. Balluchi, L. Benvenuti, M. Di Benedetto, C. Pinello, and A. Sangiovanni-Vicentelli. Automotive engine control and hybrid systems: Challenges and opportunities. *Proceedings of the IEEE*, 88(7):888–912, July 2000.
7. G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide.* Addison Wesley, 1997.
8. A. Chutinan and B. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In *Hybrid Systems : Computation and Control*, LNCS 1569, 1999.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
10. E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.
11. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification, 11th Intl. Conf.*, LNCS 1633, pages 160–171, 1999.
12. 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, 1999.
13. S. Engell, S. Kowalewski, C. Schulz, and O. Stursberg. Continuous-discrete interactions in chemical processing plants. *Proc. of the IEEE*, 88(7):1050–1068, 2000.
14. J. Esposito and V. Kumar. Efficient dynamic simulation of robotic systems with hierarchy. In *Intl. Conf. on Robotics and Automation*, pages 2818–2823, 2001.
15. J. Esposito, V. Kumar, and G. Pappas. Accurate event detection for simulating hybrid systems. In *Hybrid Systems : Computation and Control*, LNCS 2034, pages 204–217, 2001.
16. R. Fierro, A. Das, V. Kumar, and J. P. Ostrowski. Hybrid control of formations of robots. *Proc. Int. Conf. Robot. Automat.*, pages 157–162, 2001.
17. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
18. T.A. Henzinger, P. Ho, and H. Wong-Toi. HyTech: the next generation. In *Proc. TACAS'95*, LNCS 1019, pages 41–71, 1995.
19. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
20. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
21. G. Lafferriere, G. Pappas, and S. Yovine. Symbolic reachability computation for families of linear vector fields. *Journal of Symbolic Computation*, 2001.
22. K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Springer Intl. Journal of Software Tools for Technology Transfer*, 1, 1997.
23. E.A. Lee. What's ahead for embedded software. *IEEE Computer*, pages 18–26, September 2000.
24. 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.
25. 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, 1991.
26. K. McMillan. *Symbolic model checking: an approach to the state explosion problem.* Kluwer Academic Publishers, 1993.
27. R. Milner. *A Calculus of Communicating Systems.* LNCS 92, 1980.
28. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th Intl. Conf. on Computer Aided Verification*, LNCS 1254, 1997.
29. C. Tomlin, G. Pappas, and S. Sastry. Conflict resolution for air traffic management: A study in multi-agent hybrid systems. *IEEE Trans. Automatic Control*, 43(4):509–521, 1998.