Hierarchical Modeling and Analysis of Embedded Systems

Rajeev Alur, Thao Dang, Joel Esposito, Yerang Hur, Franjo Ivančić, Vijay Kumar, Insup Lee, Pradyumna Mishra, George J. Pappas, and Oleg Sokolsky

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.**

Keywords— **Hybrid Systems, Modular Design, Formal Analysis Tools, Embedded Control Systems.**

I. 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 [1]). 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 [3], aircrafts [4], and chemical process control systems [5]. Model-based design paradigm is particularly attractive because of its promise for greater design automation and formal guarantees of reliability.

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. Hybrid modeling combines these two approaches and is natural for specification of embedded systems.

We have been developing a modeling language, CHARON, that is suitable for specification of interacting embedded systems as communicating agents. CHARON has been used in the modeling and analysis of a wide range of hybrid systems, such as automotive powertrain, vehicle-to-vehicle control systems [6], biological cells [7] multi-agent systems [8], [9], infusion pump and inverted pendulum systems [10]. The two salient aspects of CHARON are that it supports modular specifications and that it has a well-defined formal semantics.

Hierarchical, modular modeling. Modern software design paradigms promote *hierarchy* as one of the key constructs for structuring complex specifications. They 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 [11]. CHARON supports both architectural and behavioral hierarchies.

Early formal models for hybrid systems include phase transition systems [12] and hybrid automata [13]. While modularity in hybrid specifications has been addressed in languages such as hybrid I/O automata [14], 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.

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 [15] and CCS [16], and is a prerequisite for developing modular reasoning principles such as compositional model checking and systematic de-

University of Pennsylvania, Department of Computer and Information Science, 200 South 33rd Street, Philadelphia, PA 19104, USA, http://www.seas.upenn.edu/hybrid/

Supported in part by DARPA MoBIES grant F33615-00-C-1707, ARO DAAD19-01-1-0473, NSF CCR-9970925, NSF CCR-9988409, NSF CCR-0086147, NSF CISE-9703220, NSF ITR CCR01-21431, and ONR N00014-97-1-0505.

sign principles such as stepwise refinement.

There are two aspects of CHARON that make it difficult to adopt existing techniques. First, the global nature of time makes it challenging to define semantics of hybrid components in a modular fashion. Second, features such as group transitions, exceptions, and history retention supporting rich hierarchical specifications cause additional difficulties. The compositional semantics of CHARON supports observational trace semantics for both modes and agents [17]. 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.

The remaining paper is organized as follows. Section II gives a short overview of related work. In Section III, we present the features of the language CHARON, and in Section IV we describe the formal semantics and accompanying compositional refinement calculus. We use examples from the automotive experimental platform of DARPA's MoBIES project for illustrative purposes. Section V gives an overview of ongoing research on formal analysis, and we conclude in Section VI with a summary of the CHARON design toolkit.

II. BACKGROUND

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 [18]. 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.

Tool support for control system design. 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 simulation. Tools such as SIMULINK recently augmented the continuous modeling with state-machine-based modeling of discrete control.

Modeling languages for hybrid systems. To benefit from object-oriented design, several languages that support object-oriented modeling of complex dynamical systems have been proposed. Omola [19], Dymola [20], and Modelica [21] provide non-causal models; that is, there is no notion of causality in the equations in the models. Those three have been used mostly for describing physical objects, whereas SHIFT [22] is more like a programming language and has been used extensively to specify automated vehicle highway systems. PTOLEMY II [23] supports the modeling, simulation, and design of concurrent systems. It incorporates a number of models of computation (such as synchronous/reactive systems, communicating sequential processes (CSP), finite state machines, continuous time, etc) with semantics that allow domains to interoperate.

All the above languages were proposed for modeling and simulation purposes and have not been used for formal verification of systems. CHARON has compositional formal semantics required to reason about systems in a modular way while incorporating many features of the aforementioned languages. Two features that are not supported by CHARON are model inheritance and dynamic creation of model instances.

Model checking. Inspired by the success of model checking in hardware verification and protocol analysis [24], [25], there has been increasing research on developing techniques for automated verification of hybrid (mixed discrete-continuous) models of embedded controllers [13], [26], [27], [28], [29]. The state-of-the-art computational tools for model checking of hybrid systems are of two kinds. Tools such as KRONOS [30], UPPAAL [31], and HYTECH [32] limit the continuous dynamics to simple abstractions such as rectangular inclusions (e.g. $\dot{x} \in [1, 2]$), and compute the set of reachable states exactly and effectively by symbolic manipulation of linear inequalities. On the other hand, emerging tools such as CHECKMATE [33], `d/dt` [34], and level-sets method [35], [36], approximate the set of reachable states by polyhedra or ellipsoids [37] by optimization techniques. Even though these tools have been applied to interesting real-world examples after appropriate abstractions, scalability remains a challenge.

III. MODELING LANGUAGE

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.

Case study. Throughout the paper we will use a recent case study to illustrate the modeling and analysis concepts within the proposed framework. The case study is based on the longitudinal control system for vehicles moving in an Intelligent Vehicle Highway System (IVHS) [38]. A detailed description of the system can be found in [39]. Before proceeding with the modeling of the problem, we present a brief informal description of the control system.

In the context of IVHS, vehicles travel in platoons and inside a platoon all the vehicles follow the leader. We con-

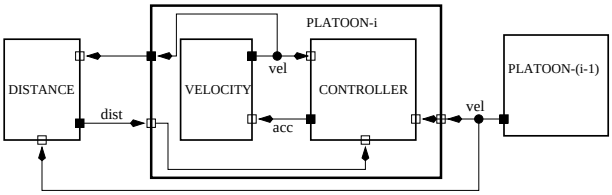


Fig. 1. The architectural hierarchy of the platoon controller

sider a platoon i and its preceding platoon $(i-1)$. Let v_i and a_i denote the velocity and acceleration of the platoon i, respectively, and d_i be its distance to the platoon $(i-1)$. The most important task of a longitudinal controller for the leader car of each platoon i is to maintain the distance d_i equal to a safety distance $D_i = \lambda_a a_i + \lambda_v v_i + \lambda_p$; in the nominal operation, $\lambda_a = 0s^2$, $\lambda_v = 1s$, and $\lambda_p = 10m$. Other tasks the controller should perform are to track an optimal velocity and trajectories for certain maneuvers. Without going into details, the controller for the leader car of platoon i proposed in [39] consists of four control laws u, which are used in different regions of the state space. These regions are defined based on the values of the relative velocity $v_i^e = 100(v_{i-1} - v_i)/v_i$ and the error between the actual and the safe inter-platoon distances $e_i = d_i - D_i$. When v_i^e and e_i change from one region to another, the control law should change accordingly. One important *property* we want to verify is that a *collision between platoons never happens*, that is, $d_i > 0m$. To this end, we consider a system with four continuous variables (d_i, v_{i-1}, v_i, a_i). The dynamics of these variables are as follows:

$$\begin{cases} \dot{d}_i & = & v_{i-1} - v_i \\ \dot{v}_{i-1} & = & a_{i-1} \\ \dot{v}_i & = & a_i \\ \dot{a}_i & = & u \end{cases} \quad (1)$$

where u is the control. One can see that the dynamics of each platoon depends on the state of its preceding platoon. We consider a pair of platoons $(i-1)$ and i and prove that the controller of the leader car of platoon i can guarantee that no collision happens regardless of the behavior of platoon $(i-1)$. More precisely, the acceleration a_{i-1} of the platoon in front is treated as *uncertain input* with values in the interval $[a_{min}, a_{max}]$ where a_{min} and a_{max} are the maximal possible deceleration and acceleration.

A. Agents and Architectural Hierarchy

The architectural hierarchy of the above platoon control system is shown in Figure 1. The agent `PLATOON-`i consists of two sub-agents, namely `VELOCITY` and `CONTROLLER`. The sub-agent `CONTROLLER` models the control laws and outputs the acceleration a_i of the platoon i. The sub-agent `VELOCITY` takes as input the variable `acc` and updates the variable `vel` of the platoon i. The agent `PLATOON-`$(i-1)$, whose role is to model all possible behaviors of the platoon in front, outputs its own velocity (variable `vel`) to the agent `PLATOON-`i. In other words, the velocity (or acceleration) of the platoon $(i-1)$ can be seen as *uncertain input* (or external disturbance) to the agent `PLATOON-`i.

Each agent has a well-defined interface which consists of its typed input and output variables, represented visually as blank and filled squares, respectively. The two variables `vel` of the agents `PLATOON-`$(i-1)$ and `PLATOON-`i are inputs to the agent `DISTANCE` which outputs the distance between the two platoons. The sub-agent `CONTROLLER` of `PLATOON-`i computes the desired acceleration a_i based on the inter-platoon distance and the velocity of the platoon in front.

Formally, an *agent*, $A = \langle TM, V, I \rangle$, consists of a set V of variables, a set I of initial states, and a set TM of modes. 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 are constructed from other agents and have many top-level modes. For example, the behavior of the agent `PLATOON-`i is given by the top-level modes of its atomic sub-agents, `VELOCITY` and `CONTROLLER`.

Figure 1 illustrates the three operations defined on agents. Agents can be *composed* in parallel with each other. The parallel agents execute concurrently and communicate through shared variables. To enable communication between the two vehicles, global variables are *renamed*. For example, variables `vel` of agents `PLATOON-`$(i-1)$ and `PLATOON-`i are renamed into `velFirst` and `velOther`, respectively, so that the agent `DISTANCE` can read them without confusion. Finally, the communication between the vehicles can be *hidden* from the outside world. In our example, only the variable `vel` is the output of a platoon agent. The variable `acc`, used internally by the agent `PLATOON-`i, cannot be accessed from the outside.

B. 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 discrete control. Each 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, through which discrete control enters and exits the mode. Entry and exit points are denoted as blank and filled circles, respectively. A top-level mode, which is activated when the corresponding agent comes into existence and is never deactivated, has a special entry point `init`.

At the lowest level of the behavioral hierarchy are atomic modes. They describe continuous behaviors. For example, Figure 2 illustrates the behavior of the mode `Track`, which specifies a control law by means of a differential constraint that asserts the relationship between desired acceleration *acc*, and input variables of the mode, representing the velocities of the platoon, the platoon in front of it, and the distance between platoons. CHARON also supports alge-

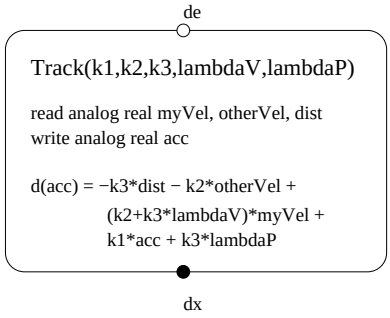


Fig. 2. Mode `Track`

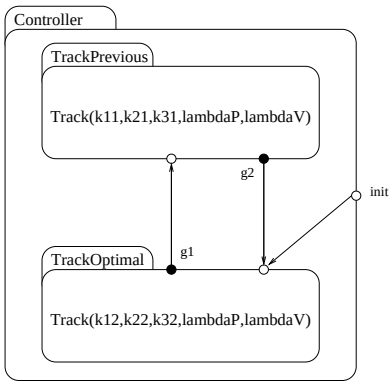


Fig. 3. Behavior of the agent `Controller`

braic constraints on variable values. In addition, an *invariant* may be used to specify how long the mode can remain active. Once an invariant is violated, the mode has to be exited by taking one of the transitions leaving the mode.

The values of `k1, k2, k3, lambdaP`, and `lambdaV` are parameters of the mode. The mode can be instantiated with different values for the parameters several times in the same model, yielding different control laws. This will be illustrated below.

Composite modes contain a number of submodes. During execution, a composite mode performs discrete transitions that connect its control points and control points of its submodes. For example, the behavior of the agent `Controller` is captured by the mode shown in Figure 3. To avoid cluttering the figure, we omit the guards on mode transitions.

Formally, a mode $M = \langle SM, V, E, X, T, Cons \rangle$ 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.

Constraints of a mode define continuous behavior of a mode in three ways. 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 values for the first derivative of x with respect to time. Additionally, only those trajectories are allowed that satisfy the invariant of the mode, which is a boolean predicate over the mode variables.

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. In the example, the entry transition of `Controller` specifies that the mode starts in the TrackOptimal submode, which will be used to "catch up" with the platoon in front. There are no exit transitions since it is a top-level mode and it has to execute forever. Every transition has a *guard*, which is a predicate over the valuations of mode variables that specifies 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 richer discrete and continuous specifications.

IV. Formal Semantics and Compositional Refinement

In this section, we first define the operational semantics of modes and agents that makes the notion of executing a CHARON model precise, and can be used, say, by a simulator. Second, we define 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 to relate different models of the same system. We can establish, for instance, that an abstract (simplified) version of a platoon refines a detailed version, and then to analyze control of platoons using the abstract version instead of the detailed one, significantly simplifying analysis. The compositional rules about refine-

ment form the basis for analysis in a system with multiple components, each with a simplified and a detailed model.

A. 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 is not active. 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 `Controller` introduced in Section III-B is illustrated in Figure 4.

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

The relation $R^C \subseteq Q_V \times \mathcal{F}_V$ gives, for every data state of the mode, the set of flows from this state. By definition, if the control state of the mode is not at dx, the set of flows for the state is empty. 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 of *macro-steps* of a mode start-

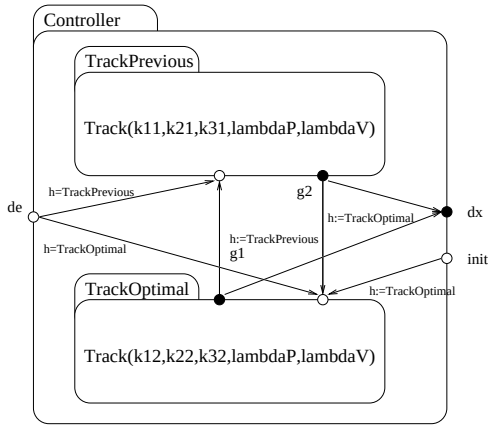


Fig. 4. The closure of a mode

ing 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 `Controller` from Figure 4. Let s be such that $h = \epsilon$ and g_1 is false. Then, there is a micro-execution for `Controller`: *init*, `TrackOptimal.`*de*, `TrackOptimal.`*dx*, *dx* (we show only the control points of the micro-execution for clarity). This means that $(s, s[h := \texttt{TrackOptimal}]) \in \mathtt{R^D_{init,dx}}$. If g_1 is true in a state s', then $(s', s'[h := \texttt{TrackPrevious}]) \in \mathtt{R^D_{init,dx}}$ corresponding to the micro-execution *init*, `TrackOptimal.`*de*, `TrackOptimal.`*dx*, `TrackOptimal.`*de*, `TrackOptimal.`*dx*, *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 $\mathcal{R}$:

$$(e_0, s_0) \xrightarrow{\lambda_1} (e_1, s_1) \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_n} (e_n, s_n). \qquad (2)$$

Trace semantics. To be able to define a refinement relation between modes, we consider 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 modular semantics in a way that such distinction is not made seems much more difficult.

B. Trace Semantics for Agents

An execution of an agent, $A = \langle TM, V, I \rangle$ 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 the agent A 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.

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.

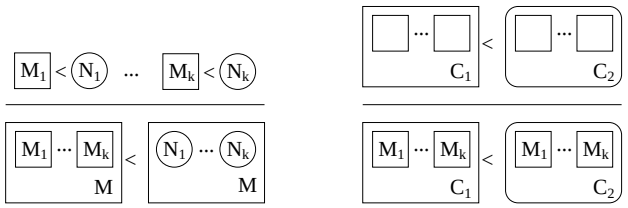


Fig. 5. Compositionality rules for modes

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.

C. Compositionality Results

As shown in [17], our semantics is compositional for both modes and agents as follows. 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 and a mode N are said to be *compatible* if $M.V_g = N.V_g$, $M.E{=}N.E$ and $M.X{=}N.X$, i.e., they have the same global variables and control points. For two compatible modes M and N, we say that M refines N, denoted $M{\preceq}N$, if $L_M{\subseteq}L_N$, i.e., every trace of M is a trace of 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 5, 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 5, right. Precise statements of the results can be found in [17].

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 $A_1||A_2 \preceq B_1||B_2$

V. Analysis

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

A. Simulation Techniques

Numerical simulation is an important tool for designing and analyzing many types of control systems, including hybrid systems. In addition to pure simulation, numerical approximation techniques are increasingly being used in reachability computations, verification and other forms of automated analysis [33], [36], [40].

All numerical simulators operate based on some assumptions about the nature of the systems being simulated. The degree to which the system adheres to these assumptions determines how accurate the results are and what computational effort is required to generate them. Traditional numerical integration techniques typically make assumptions that tend to be violated by hybrid system models.

In addition, the hierarchical structure of the models yields the following two observations. Often, high level modes have very slow changing dynamics while low-level detailed models may possess fast changing dynamics. Multiple agents in a model may be decoupled in the continuous sense, yet interact through discrete messaging. Both observations may be used to increase efficiency of simulators.

Therefore, novel simulation techniques, specific to hierarchical hybrid systems are warranted. The need for specialized simulation tools has been recognized to some degree in the literature [41], [42]. Several hybrid system simulators have been introduced (see for example Modelica [43], ABACUSS [44], 20-sim [45], SHIFT [22] and χ [46] as well as others reviewed in [42]). Most of the previous research has focused on properly detecting and locating discrete transitions, while largely ignoring the remaining issues. In this section, we describe three techniques that exploit the hierarchical structure of hybrid system models to provide increased accuracy and efficiency during simulation.

A.1 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. Formally, the *event detection problem* is posed as follows. Given a system

$$\dot{s} = \begin{cases} f^{M_1}(s), \text{ if } g(s) < 0 \\ f^{M_2}(s), \text{ if } g(s) \geq 0, \end{cases} \tag{3}$$

where the mode $M \in \{M_1, M_2\}$ and $s \in Q_{M.V_*}$ is the continuous (or data) state, one would like to simulate the

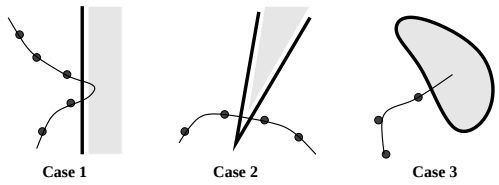


Fig. 6. Cases 1 and 2 illustrate situations in which naive simulators can fail to detect transitions by selecting integration points which completely "miss" the guard set; Case 3 depicts a situation in which even sophisticated methods fail, when the event occurs near a region where the differential equation has a singularity at which the right hand side cannot be evaluated.

flow of s according to f^{M_1} until the *first* time, t', that the event $g(s(t')) = 0$ occurs. We assume that initially $M = M_1$ meaning that f^{M_1} is the active flow. Additionally we assume that the guard $g(s) < 0$ is true initially.

It is generally agreed that any algorithm which addresses this problem should possess the following attributes:
1. the algorithm should be guaranteed to detect an event if one occurs, and guaranteed to not return false positives;
2. if more than one event occurs in a given time interval, the algorithm ought to be capable of determining and reporting the *first* event;
3. once it is determined that an event has occured, the algorithm should be able to localize precisely the time, t', at which it occured; and
4. provided all of the above criteria are fulfilled, the algorithm should be as efficient as possible.

Early event detection methods, such as [47], [48], [49], [50], lack rigor, and are not guaranteed to correctly detect an event in many situations. More recent approaches, see [51], [52] for example, satisfy the first three objectives in most situations while being reasonably efficient. However, a situation in which nearly all current simulators fall short is when switches occur near model singularities. Since the step-size selection scheme for the integration is typically independent of the event detection algorithm, it is entirely possible that the integrator will take a step into the region where $f^{M_1}(x)$ is undefined. If the particular integration method has an intermediate step which requires evaluating the derivative at this state inside the singular region, a floating point exception is generated and the simulation fails abruptly. Some of these problematic situations are illustrated in Fig. 6.

We have developed a method [53] which is guaranteed to detect enabling of all transitions, including those occuring near singular regions. We attempt to overcome this problem by treating the event detection problem as a control system design problem. We consider the continuous dynamics of the system and the numerical integration method (we use Linear Multistep Methods - see [54] for further details):

$$s_{k+1} = s_k + h\{\sum_{j=1}^{m} \beta_j f_{k-j+1}\} \tag{4}$$

as our collective dynamic system, where t_k is the time of the k^{th} simulation step, s_k is the value of the state at t_k, $h = t_{k+1} - t_k$ is the simulation step size, and $\sum_{j=1}^{m} \beta_j f_{k-j+1}$ is some weighted combination of past values of the derivative which approximates the flow on $[t_k, t_{k+1}]$.

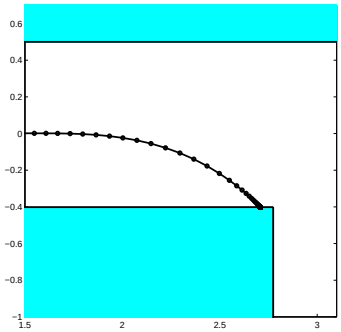


Fig. 7. The simulation takes successively smaller steps to properly locate the point at which the vehicle clips the corner.

Returning to our control system analogy, the integration step size h is treated as an input and the value of the transition guard, $g_k = g(s_k)$, or switching function is the output. The task at hand is to integrate the ordinary differential equation (ODE) until the boundary of the guard set is reached, taking care to never evaluate the right hand side of the ODE inside the guard set. In terms of our control system analogy the problem can be rephrased as: design a feedback law that zeros the output with no overshoot. The resulting solution is essentially an Input/Output Linearization in discrete time. For a linear guard the output dynamics would be

$$g_{k+1} = g_k + h\frac{\partial g}{\partial s}\{\sum_{j=1}^{m} \beta_j f_{k-j+1}\} \tag{5}$$

selecting the step size, h, as

$$h = \frac{(\gamma - 1)g_k}{\frac{\partial g}{\partial s}\{\sum_{j=1}^{m} \beta_j f_{k-j+1}\}} \tag{6}$$

results in equation (5) appearing as $g_{k+1} = \gamma g_k$. By selecting the constant $0 < \gamma < 1$ we are ensured $g_k \to 0$ while maintaining $g_k \leq 0$. Thus the simulation settles to the transition surface without overshooting it and crossing into the singular region. This technique is illustrated in Fig. 7 where a vehicle is trying to go around a corner and the simulation must detect if it clears the corner. One can see how the simulation converges onto the exact point at which the collision occured.

A.2 Multi-Rate 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 slowly 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 methods [55], [56] is to use larger

step sizes for the slow changing sets of differential equations and smaller step sizes for the differential equations evolving on the fast time scale. Such a strategy increases efficiency without compromising accuracy. Areas of application include simulating integrated circuits and molecular and stellar dynamics [57], [58], [59]. Despite the seemingly natural connection, they have never previously been used in hierarchical hybrid systems simulation. In [60] we introduce a multi-rate algorithm for simulating hierarchical hybrid systems.

A.3 Multiagent Simulation

Multiagent hybrid systems are sets of interacting hybrid systems. In the case of the automated highway example, each vehicle may be modeled as an individual agent, however one may like to consider the dynamics of an entire group of vehicles collectively to see how they interact. The continuous dynamics of each vehicle is physically decoupled from that of the other agents and typically they operate independently. However certain important discrete events may depend on the state of two or more agents. Examples of this would be when two cars come dangerously close, one car informs a group of vehicles that it is merging into the platoon, etc. Most multi-agent systems of this form, when modeled in CHARON, have the following mathematical structure

$$\dot{x} = f_x(x) \quad (7)$$

$$\dot{y} = f_y(y) \quad (8)$$

$$g(x,y) \leq 0 \quad (9)$$

where x and y are the continuous states of agent 1 and agent 2, their dynamics are given by the flows $f_x : \mathbb{R}^n \to \mathbb{R}^n$ and $f_y : \mathbb{R}^m \to \mathbb{R}^m$, and the predicate $g(x,y) < 0$ guards a transition for one or both agents. Note that each agent's ODE's are decoupled; however, coupling is introduced through the guards.

From the point of view of simulating the continuous dynamics, it is not necessary to synchronize the integration rates of the two cars since they are decoupled. Each set of ODE's should maximize the trade-off between accuracy and efficiency by selecting the largest possible integration step size which is able to recreate that agents' dynamics within some acceptable user-specified error tolerance. Unfortunately, properly detecting the occurrence of events, $g(x,y) = 0$, requires that the value of the state be reported in a synchronized fashion. Traditionally simulators compute the best step size for each agent, and then take the minimum as a global step size. This can result in significant inefficiencies.

Our goal is to simulate each agent with a different step size while still ensuring proper event handling. The idea is to allow the simulation for each agent to proceed independently when no events are about to occur. Only when events seem likely do we adaptively select the step sizes to bring all of the agents into synchronization to properly detect the event.

In the case of N-agents our approach to this problem, reported in [61], is to define N local clocks, $t_1, \ldots t_N$ and

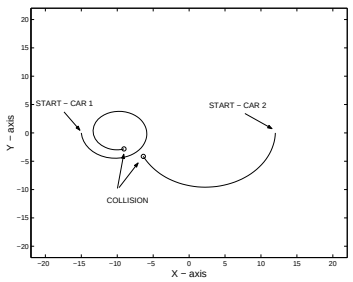


Fig. 8. The trajectories of the two cars in the plane.

N step sizes $h_1, \ldots, h_N$, one for each agent. The step sizes are selected based on the system dynamics in such a way as to simultaneously synchronize the local clocks and detect the event using the control theoretic technique of Input-Output Linearization.

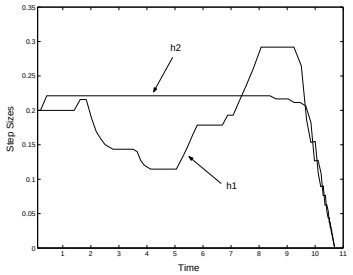


Fig. 9. Step sizes used for the two cars depicted in Figure 8. The step sizes h_1 and h_2 are selected independently away from the constraint but are brought into synchronization when an event is impending.

Figures 8 and 9 illustrate how the simulation for two agents might proceed. Figure 8 shows the trajectories of the two cars. The simulation tries to detect when the cars collide. Figure 9 displays how the step sizes are selected independently throughout most of the simulation.When the system approaches an event, the local clocks automatically synchronize.

A.4 Distributed Simulation

The main idea behind distributed simulation is to get speedup by utilizing multiple computing resources since simulations of complex systems are normally very slow. Distributed simulation techniques are categorized as conservative or optimistic based on how local clocks are synchronized. If the local clock of the agent always advances and does not go backward, it is called *conservative simulation*; otherwise, it is called *optimistic simulation*. Conservative simulation techniques ensure that the local clock l_c of the agent either advances or stops, but does not rollback. In optimistic simulation, the focus is to exploit possible parallelism as much as possible by allowing each agent to run at a different speed without the guarantee that no event occurs between t_1 and t_2 when its local clock l_c is advanced from t_1 to t_2. If an event e that occurred at time t_e gets recognized by the agent at t_r, where $t_r > t_e$, the simulator provides a rollback operation by restoring the local clock l_c to an earlier time such that the event e can be

handled if and when it occurs. Note that the event e may not occur at all if rollbacks are propagated to other agents in such a way that the event e becomes no longer possible.

Our approach to simulate hybrid systems in a distributed fashion is to utilize more computing resources by exploiting inherent modularity of systems described in CHARON. By modularity, we mean two things. One is behavioral modularity captured by mode and the other is architectural modularity by agent. One way to exploit mode-level modularity within a single agent is to use multiple rates for the simulation of the same agent as described in V-A.2. Another way is to distribute atomic agents to exploit agent-level modularity. When the agents are distributed, they need to synchronize to update their states as the agents share information. Here, the challenge is how to reduce synchronization overheads among distributed agents. We briefly describe our conservative algorithm and optimistic algorithms.

In a conservative approach, we decompose functions into sub-functional blocks and the simulator allows the agent to execute the next block only when all the agents complete the current block. Although our conservative approach allows to simulate hybrid systems, the disadvantage is that overhead resulting from communications degrades the possible performance gain from distributing computations. Thus, we can get speedup only in simulating very computation-oriented hybrid systems. Our optimistic simulation algorithms are to address the overhead problems. The main features of the algorithms are as follows. First, to reduce communication overhead, we let agents synchronize just before the new value of a shared variable is necessary instead of communicating every update round. Second, to reduce computation overhead due to numerical integration, we simulate the agent with its approximated polynomial dynamics and resolve the possible misses of events with a rollback operation. This allows each agent to execute its computation without integrating the shared variables controlled by other agents. Our approach is *optimistic* in the sense that each agent goes forward even when there is no guarantee that their clocks do not have to go backward.

A.5 Case Study

We now consider simulation of the platoon controller under normal conditions. Figures 10-12 are snapshots of the CHARON plotter and show the simulation results for the following scenario. Initially, the distance between the two platoons is large, and the platoon i is moving faster than the platoon in front $(i-1)$ and is therefore closing the gap. We let the velocity of the platoon in front be a sinusoidal function of time starting at an initial value 20. One can see from the figures that the controller of platoon i, initially in the mode "track optimal velocity", first decreases the gap between the two platoons by accelerating. When its distance to the preceding platoon becomes small, the controller slowly decelerates and switches to mode "track velocity of previous car" approximately at time 8.2. The controller then tries to follow the platoon in front at some constant distance. More simulation trace plots of this example can be found in [6].

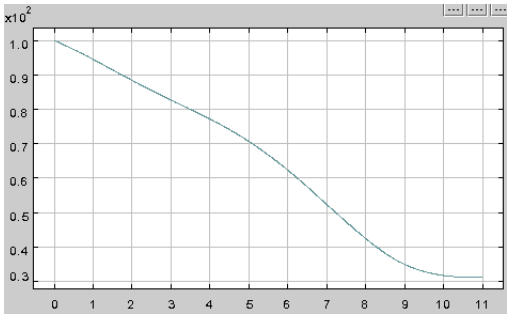


Fig. 10. The distance d_i between the two platoons.

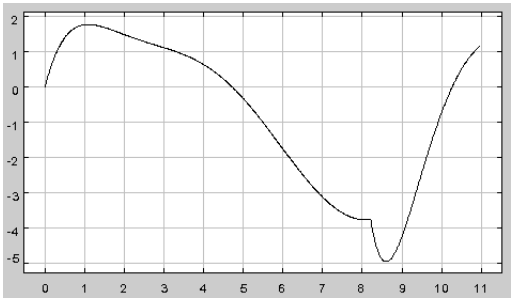


Fig. 11. The acceleration a_i of the platoon i.

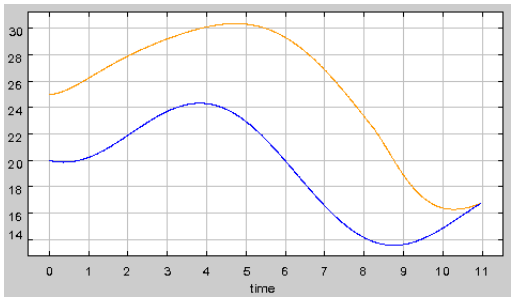


Fig. 12. The velocity of the platoon i and the preceding platoon $(i-1)$ (the platoon i moves faster).

B. State-Space Exploration Techniques

B.1 Exact Reachability using Requiem

Formal verification of safety requirements of hybrid systems requires the computation of reachable states of continuous systems. **Requiem** is a Mathematica package which, given a nilpotent linear differential equation and a set of initial conditions, symbolically computes the exact set of reachable states. 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 reals as an ordered field [62]. Given a nilpotent system and a set defined by polynomials inequalities, **Requiem** automatically generates the quantifier elimination problem and invokes the quantifier elimination package in Mathematica

4.0. If the computation terminates, it returns the quantifier free formula describing the reachable set. More details can be found in [62]. The entire package is available at `www.seas.upenn.edu/hybrid/requiem.html`.

Parametric analysis using Requiem. We demonstrate the use of **Requiem** on the platoon controller described earlier. The experimental nature of the current quantifier elimination package makes it impossible to apply it to the system described by Equation (1). We thus simplify the controller with equivalent dynamics, which controls the acceleration of the platoon i instead of its derivative. This approximation results in the three dimensional system described by

$$\begin{cases} \dot{d}_i &= v_{i-1} - v_i \\ \dot{v}_{i-1} &= a_{i-1} \\ \dot{v}_i &= u \end{cases} \tag{10}$$

We treat the acceleration of the preceding platoon a_{i-1} as a parametric disturbance and control the acceleration $\dot{v}_i$ of the following platoon. The problem is to find the set of conditions on the parameter set $\{a_{i-1}, b, c\}$ and the state variables, which would lead to a collision ($d_i \leq 0$) when we apply a control u of the form $bt + c$ where b and c are integer constants. We use **Requiem**'s parametric backward reachability function to obtain the quantifier free formula. By giving specific values to the parameters and initial conditions, we can see whether the formula reduces to *true* or *false*. For example, we can prove the expected result that when the vehicles are started close to each other ($d = 1$) and the control parameters b and c are positive, collision is unavoidable, whereas if b and c are negative, collision does not occur. The entire example and the output is available at `www.seas.upenn.edu/hybrid/requiem/ReqIEEE.html`.

B.2 Predicate Abstraction

In the world of program analysis, predicate abstraction has emerged to be a powerful and popular technique for extracting finite-state models from complex, potentially infinite state, discrete systems (see [63], [64], [65], [66] for a sampling of this active research area). A verifier based on this scheme requires three inputs, the (concrete) system to be analyzed, the property to be verified, and a finite set of boolean predicates over system variables to be used for abstraction. An abstract state is a valid combination of truth values to the boolean predicates, and thus, corresponds to a set of concrete states. There is an abstract transition from an abstract state A to an abstract state B, if there is a concrete transition from some state corresponding to A to some state corresponding to B. The job of the verifier is to compute the abstract transitions, and to search in the abstract graph looking for a violation of the property. If the abstract system satisfies the property, then so does the concrete system. If a violation is found in the abstract system, then the resulting counter-example can be analyzed to test if it is a viable execution of the concrete system. This approach, of course, does not solve the verification problem by itself. The success crucially depends on the ability to identify the "interesting" predicates, either manually or by some automated scheme, and on the ability of the verifier to compute abstract transitions efficiently. Nevertheless, it has led to opportunities to bridge the gap between code and models and to combine automated search with user's intuition about interesting predicates. Tools such as Bandera [67], SLAM [68], and Feaver [69] have successfully applied predicate abstraction for analysis of C or Java programs.

Inspired by this trend, we develop algorithms for invariant verification of hybrid systems using discrete approximations based on predicate abstractions. Consider a hybrid automaton with n continuous variables and a set L of locations. Then the continuous state-space is $L \times \mathbb{R}^n$. For the sake of efficiency, we restrict our attention where all invariants, switching guards, and discrete updates of the hybrid automaton are specified by linear expressions, and the continuous dynamics is linear, possibly with uncertain, bounded input. For the purpose of abstraction, the user supplies initial predicates $p_1 \ldots p_k$, where each predicate is a polyhedral subset of $\mathbb{R}^n$. In the abstract program, the n continuous variables are replaced by k discrete boolean variables, one boolean variable b_i for each predicate p_i. A combination of values to these k boolean variables represents an abstract state corresponding to a set of continuous states, and the abstract state space is $L \times \mathbb{B}^k$. Our verifier performs an on-the-fly search of the abstract system by symbolic manipulation of polyhedra.

The core of the verifier is the computation of the transitions between abstract states that capture both discrete and continuous dynamics of the original system. Computing discrete successors is relatively straightforward, and involves computing weakest preconditions, and checking non-emptiness of an intersection of polyhedral sets. The implementation attempts to reduce the number of abstract states examined by exploiting the fact that each abstract state is an intersection of k linear inequalities. For computing continuous successors of an abstract state A, we use a strategy inspired by the techniques used in CheckMate [33] and `d/dt`[34]. The basic strategy computes the polyhedral slices of states reachable from A at fixed times $r, 2r, 3r, \ldots$ for a suitably chosen r, and then, takes the convex-hull of all these polyhedra to over-approximate the set of all states reachable from A. However, while tools such as CheckMate and `d/dt` are designed to compute a "good" approximation of the continuous successors of A, we are interested in testing if this set intersects with a new abstract state. Consequently, our implementation differs in many ways. For instance, it checks for nonempty intersection with other abstract states of each of the polyhedral slices, and omits steps involving approximations using orthogonal polyhedra and termination tests (see [34]).

Postulating the verification problem for hybrid systems as a search problem in the abstract system has many benefits compared to the traditional approach of computing approximations of reachable sets of hybrid systems. First, the expensive operation of computing continuous successors is applied only to abstract states, and not to inter-

mediate polyhedra of unpredictable shapes and complexities. Second, we can prematurely terminate the computation of continuous successors whenever new abstract transitions are discovered. Finally, we can explore with different search strategies aimed at making progress in the abstract graph. For instance, our implementation always prefers computing discrete transitions over continuous ones. Our early experiments indicate that improvements in time and space requirements are significant compared to a tool such as `d/dt`. A more detailed description of our predicate abstraction technique for hybrid systems can be found in [70].

Verification of the platoon controller using predicate abstraction. To formally prove the safety property of this longitudinal controller, we make use of the reachability method using predicate abstraction. Here, we focus only on two regions which are critical from a safety point of view: "track optimal velocity" ($v_i^e \leq -10$ and $e_i \geq -1m - \epsilon$) and "track velocity of previous car" ($v_i^e \leq -10$ and $e_i \leq -1m$). We include a thickening parameter $\epsilon > 0m$ into the model to add non-determinism to it. The two regions under consideration overlap allowing the controller to either use the "track optimal velocity" controller or the "track velocity of previous car" controller in this ϵ-thick region. Besides adding some non-determinism to the model, it also provides improved numerical stability to the simulation and reachability computation, as it is numerically hard to determine the exact time at which a switch occurs.

The respective control laws u_1 and u_2 are as follows:

$$u_1 = \frac{1}{8}d_i + \frac{3}{4}v_{i-1} - (\frac{3}{4} + \frac{1}{8}\lambda_v)v_i - \frac{3}{2}a_i - \frac{1}{8}\lambda_p \quad (11)$$

$$u_2 = d_i + 3v_{i-1} - (3 + \lambda_v)v_i - 3a_i - \lambda_p. \quad (12)$$

Note that these regions correspond to situations where the platoon in front moves considerably slower and, moreover, the second region is particularly safety critical because the inter-platoon distance is smaller than desired.

To construct the discrete abstract system, in addition to the predicates of the invariants and guards we include some predicates over the distance variable to be able to separate the bad region from the reachable set: $d_i \leq 0, d_i \geq 2, d_i \geq 10, d_i \geq 20$. The total number of initial predicates is 11. For the initial set specified as $20 \leq d_i \leq 100 \wedge 15 \leq v_{i-1} \leq 18 \wedge 20 \leq v_i \leq 25$, the tool found 14 reachable abstract states and reported that the system is safe. Note this property has been proven in [71] using optimal control techniques for individual continuous modes without mode switches. Here, we prove the property for all possible behaviors of the controller.

VI. 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 [72], and the plotter uses a package from the modeling tool Ptolemy [23].

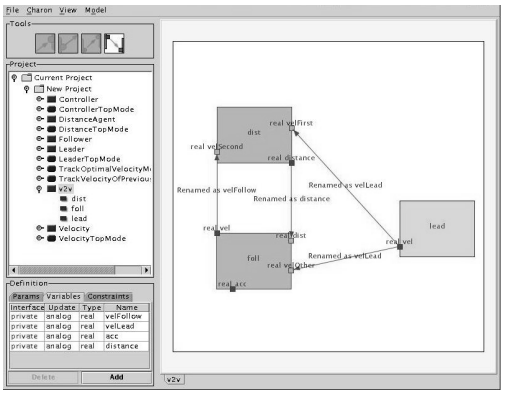


Fig. 13. The visual input tool of Charon. The arrows depict variable renamings.

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 at a given level of hierarchy. The visual input tool is depicted in Figure 13, showing one level of the platoon controller from Figure 1. By clicking on the sub-agents, the user can explore the lower levels of hierarchy. The interpreter of the visual input translates the specification into text-based Charon source code using an intermediate XML-based representation.

Once a set of edited and saved Charon language files 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 14, which displays the same model as Figure 13.

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 files.

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 15, resembles concepts

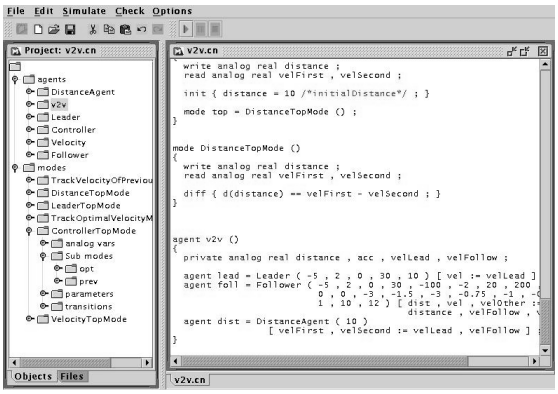


Fig. 14. The editor frame on the right hand side of the CHARON desktop and the corresponding project frame on the left.

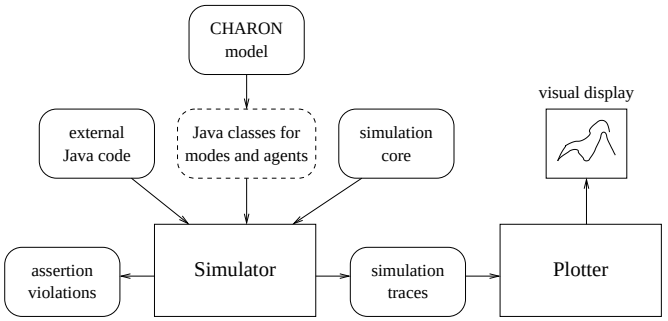


Fig. 15. The simulation methodology of CHARON

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 simulator. Given a set of CHARON files, 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. The simulation results obtained in Figures 10-12 have been produced using the CHARON plotter.

In addition, the simulator checks *assertions* that are placed in the CHARON model by the user. Assertions can be added to any mode or agent in the model. They are state predicates over the variables of the mode or agent and are supposed to be true whenever the mode is active or, for agents, always. If an assertion is violated during a simulation, the simulator stops and the trace produced by the simulator can be used to find the source of the violation.

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

ACKNOWLEDGMENTS

We would like to thank Dan Huber and Michael McDougall for their work on the CHARON visual interface, Usa Sammapun for her contribution to the simulator generator, and Valentina Sokolskaya for the implementation of the type-checker. In addition, we would like to thank Rafael Fierro and Radu Grosu for their various contributions during the initial development of CHARON.

REFERENCES

[1] E.A. Lee. What's ahead for embedded software. *IEEE Computer*, pages 18–26, September 2000.

[2] R. Alur, J. Esposito, M. Kim, V. Kumar, and I. Lee. Formal modeling and analysis of hybrid systems: A case study in multi-robot coordination. In *Proceedings of the Formal Methods '99*, LNCS 1708, pages 212–232. Springer, 1999.

[3] 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.

[4] C. Tomlin, G. J. Pappas, and S. Sastry. Conflict resolution for air traffic management: A study in muti-agent hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):509–521, 1998.

[5] S. Engell, S. Kowalewski, C. Schulz, and O. Stursberg. Continuous-discrete interactions in chemical processing plants. *Proceedings of the IEEE*, 88(7):1050–1068, July 2000.

[6] F. Ivančić. Report on verification of the MoBIES vehicle-vehicle automotive OEP problem. Technical Report MS-CIS-02-02, University of Pennsylvania, March 2002.

[7] R. Alur, C. Belta, F. Ivančić, V. Kumar, M. Mintz, G. Pappas, H. Rubin, and J. Schug. Hybrid modeling and simulation of biomolecular networks. In M.D. Di Benedetto and A. Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control, Fourth International Workshop*, LNCS 2034, pages 19–32. Springer-Verlag, March 2001.

[8] E. Aaron, F. Ivančić, and D. Metaxas. Hybrid models of navigation strategies for games and animations. In C. Tomlin and M.R. Greenstreet, editors, *Hybrid Systems: Computation and Control, Fifth International Workshop*, LNCS 2289, pages 7–20. Springer-Verlag, March 2002.

[9] E. Aaron, F. Ivančić, O. Sokolsky, and D. Metaxas. A framework for reasoning about animation systems. In *Third International Workshop on Intelligent Virtual Agents*, LNCS 2190, pages 47–60, 2001.

[10] R. Fierro, Y. Hur, I. Lee, and L. Sha. Modeling the Simplex Architecture using CHARON. In *Proceedings of the 21st IEEE Real-Time Systems Symposium WIP Sessions*, pages 77–80, 2000.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] R. Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer-Verlag, 1980.

[17] 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. Springer-Verlag, 2001.

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

[19] S. E. Mattsson and M. Anderson. The ideas behind omola. In *CACSD 92: IEEE Symposium on Computer Aided Control System Design*, pages 23–29, 1992.

[20] H. Elmqvist, F. E. Cellier, and M. Otter. Object-oriented mod-

eling of hybrid systems. In *Proceedings of European Simulation Symposium*, pages 31–41, 1993.

[21] H. Elmqvist, S. E. Mattsson, and M. Otter. Modelica – The new object-oriented modeling langugae. In *Proceedings of the 12th European Simulation Multiconference*, pages 127–131, 1998.

[22] A. Deshpande, A. Gollu, and L. Semenzato. The shift programming language and run-time system for dynamic networks of hybrid automata. Technical Report UCB-ITS-PRR-97-7, University of California at Berkeley, 1997.

[23] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Overview of the ptolemy project. Technical Report UCB/ERL M99/37, University of California at Berkeley, 1999.

[24] E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[25] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[26] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[27] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 2000.

[28] N. Halbwachs, Y. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *International Symposium on Static Analysis*, LNCS 864, 1994.

[29] T.A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pages 278–293, 1996.

[30] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, LNCS 1066, pages 208–219. Springer-Verlag, 1996.

[31] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1, 1997.

[32] T.A. Henzinger, P. Ho, and H. Wong-Toi. HyTech: the next generation. In *TACAS 95: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1019, pages 41–71. Springer-Verlag, 1995.

[33] A. Chutinan and B.K. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In *Hybrid Systems: Computation and Control, Second International Workshop*, LNCS 1569, pages 76–90, 1999.

[34] E. Asarin, O. Bournez, T. Dang, and O. Maler. Approximate reachability analysis of piecewise-linear dynamical systems. In *Hybrid Systems: Computation and Control, Third International Workshop*, LNCS 1790, pages 21–31, 2000.

[35] M. Greenstreet and I. Mitchell. Reachability analysis using polygonal projections. In *Hybrid Systems: Computation and Control, Second International Workshop*, LNCS 1569, pages 103–116, 1999.

[36] I. Mitchell and C. Tomlin. Level set methods for computation in hybrid systems. In *Hybrid Systems: Computation and Control, Third International Workshop*, volume LNCS 1790, pages 310–323, 2000.

[37] A. Kurzhanski and P. Varaiya. Ellipsoidal techniques for reachability analysis. In *Hybrid Systems: Computation and Control, Third International Workshop*, LNCS 1790, pages 202–214, 2000.

[38] P. Varaiya. Smart cars on smart roads: problems of control. *IEEE Trans. Automatic Control*, 38(2), 1993.

[39] D. Godbole and J. Lygeros. Longitudinal control of a lead card of a platoon. *IEEE Transactions on Vehicular Technology*, 43(4):1125–1135, 1994.

[40] T. Dang and O. Maler. Reachability analysis via face lifting. In T. Henzinger and S. Sastry, editors, *Hybrid Systems: Computation and Control*, volume 1386 of *Lecture Notes in Computer Science*, pages 96–109, Berlin, 1998. Springer Verlag.

[41] S. Kowalewski, M.Fritz, H. Graf, J.Preubig, S.Simon, O.Stursberg, and H.Treseler. A case study in tool-aided analysis of discretely controled continuous systems: the two tanks problem. In *Hybrid Systems V*, volume 1567 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[42] P.Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In F.W. Vaandrager and J. H. van Schuppen, editors, *Hybrid Systems : Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*, pages 163–177. Springer Verlag, 1999.

[43] P. Fritzson and V. Engelson. Modelica - a unified object-oriented language for system modelling and simulation. In *ECOOP 98: The 12th European Conference on Object-Oriented Programmi ng*, pages 67–90, 1998.

[44] R. Allgor P. Barton and W.Feehery. A large scale differential-algebraic and parametric sensitivity solver. *ABACUSS project report*, 97(1):1–4, 1997.

[45] J. Broenink. Modeling, simulation and analysis with 20-sim. *Journal A. Special issue on CACSD*, 38(3):22–25, 1995.

[46] D. van Beck G. Fabian and J. Rooda. Integration of the discrete and the continuous behavior in the hybrid chi simulator. In *Proceedings 1998 european simulation multiconference*, pages 252–257, 1998.

[47] M.B. Carver. Efficient integration over discontinuities in ordinary differential equation simulations. *Mathematics and Computers in Simulation*, XX:190–196, 1978.

[48] F. Cellier. *Combined discrete/ continuous system simulation by use of digital computers: techniques and tools*. PhD thesis, ETH Zurich, Zurich, Switzerland, 1979.

[49] C.W. Gear and O.Osterby. Solving ordinary differential equations with discontinuities. Technical report, Dept. of Comput. Sci., University of Illinois, 1981.

[50] L.F.Shampine, I.Gladwell, and R.W.Brankin. Reliable solution of special event location problems for ODEs. *ACM transactions on Mathematical Software*, 17(1):11–25, March 1991.

[51] V. Bahl and A. Linninger. Modeling of continuous–discrete processes. In *Hybrid Systems : Computation and Control*, LNCS 2034, pages 387–402. Springer-Verlag, 2001.

[52] T.Park and P.Barton. State event location in differential-algebraic models. *ACM transactions on modeling and computer simulation*, 6(2):137–165, 1996.

[53] 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. Springer-Verlag, 2001.

[54] U. Ascher and L.Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998.

[55] C.W.Gear and D.R.Wells. Multirate linear multistep methods. *BIT*, 24:484–502, 1984.

[56] R.C.Rice. Split Runge-Kutta methods for simultaneous equations. *Journal of Res. National Bureau of Standards*, 64B:151–170, 1960.

[57] C.Engstler and C.Lubich. Multirate extrapolation methods for differential equations with different time scales. *Computing*, 58:173–185, 1996.

[58] M.Gunther and P.Rentrop. Multirate row methods and latency of electrical circuits. *Applied Numerical Mathematics*, 13:83–102, 1993.

[59] J. Sand and S. Skelboe. Stability of backward euler multirate methods and convergence of waveform relaxation. *BIT*, 32:350–366, 1992.

[60] J. Esposito and V. Kumar. Efficient dynamic simulation of robotic systems with hierarchy. In *IEEE International Conference on Robotics and Automation*, pages 2818–2823, May 2001.

[61] J. Esposito, G. Pappas, and V. Kumar. Multi-agent hybrid system simulation. IEEE Conference on Decision and Control, December 2001.

[62] G. Lafferriere, G. Pappas, and S. Yovine. Symbolic reachability computation for families of linear vector fields. *Journal of Symbolic Computation*, 32(3):231–253, September 2001.

[63] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[64] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[65] S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification, 11th International Conference*, LNCS 1633, pages 160–171. Springer, 1999.

[66] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[67] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of 22nd Inter-*

national Conference on Software Engineering, pages 439–448, 2000.

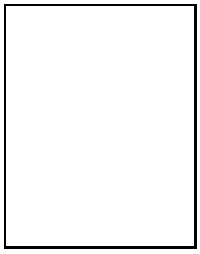
[68] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop on Model Checking of Software*, LNCS 1885, pages 113–130. Springer, 2000.

[69] G.J. Holzmann and M.H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.

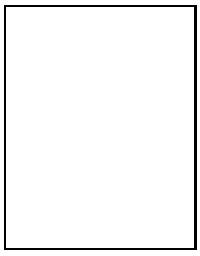
[70] R. Alur, T. Dang, and F. Ivančić. Reachability analysis of hybrid systems via predicate abstraction. In C. Tomlin and M.R. Greenstreet, editors, *Hybrid Systems: Computation and Control, Fifth International Workshop*, LNCS 2289, pages 35–48. Springer-Verlag, March 2002.

[71] A. Puri and P. Varaiya. Driving safely in smart cars. Technical Report UBC-ITS-PRR-95-24, California PATH, University of California in Berkeley, July 1995.

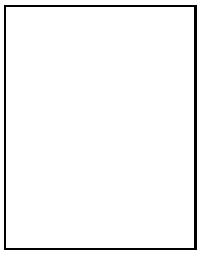
[72] R. Alur, L. de Alfaro, R. Grosu, T.A. Henzinger, M. Kang, R. Majumdar, F. Mang, C.M. Kirsch, and B.Y. Wang. MOCHA: A model checking tool that exploits design structure. In *Proceedings of 23rd International Conference on Software Engineering*, pages 835–836, 2001.



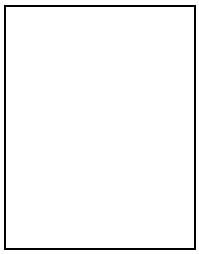
Rajeev Alur is a Professor of Computer and Information Science at the University of Pennsylvania. He received his B.Tech. degree in Computer Science from the Indian Institute of Technology at Kanpur, and his Ph.D. degree in Computer Science from Stanford University. Before joining the Penn faculty, Rajeev spent six years at Bell Laboratories, Lucent Technologies. His research interests include software engineering, design automation for embedded systems, and applied formal methods. He has published more than 70 articles in refereed journals and conference proceedings, and served on numerous scientific committees. He is well known for his research on timed and hybrid automata, a framework for specification and analysis of real-time systems. He co-organized and co-chaired the 1995 Workshop on Hybrid Systems, and the 1996 Conference on Computer-Aided Verification. He serves on the editorial board of the journal Formal Methods in System Design (Kluwer), and has won awards such as the Sloan Faculty Fellowship and the NSF CAREER award.



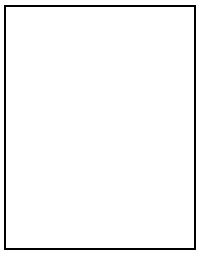
Thao Dang received her Diplôme d'Ingénieur and her M.Sc. in Electrical Engineering in 1996 from Ecole Nationale Supérieure d'Ingénieurs Electriciens of Grenoble. She received her Ph.D. in Automatic Control in 2000 from the VERIMAG laboratory in Grenoble. In 2001 she worked as a postdoctoral research associate at the Department of Computer and Information Science of the University of Pennsylvania. Since 2002, she is a research scientist at the CNRS (French National Center of Scientific Research) and a member of the VERIMAG laboratory. Her research interests are modeling, verification and control of hybrid systems and their applications in design and analysis of embedded real-time systems.



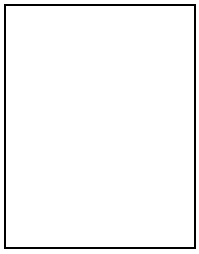
Joel Esposito is a Doctoral Candidate at the University of Pennsylvania's General Robotic and Active Sensory Perception (GRASP) Lab. His research interests include simulation of hybrid and embedded systems as well as developing other numerical and computational tools for control system design. Joel also works on motion planning and control for mobile robotics.



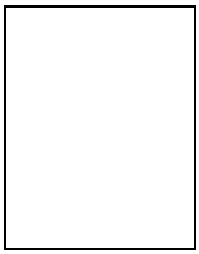
Yerang Hur received his B.S. and M.S. in Computer Engineering in 1994 and 1996, respectively from Seoul National University Korea. He is currently a Ph.D. candidate of the Department of Computer and Information Science at the University of Pennsylvania. His research interest is in the area of design automation, modeling and analysis of hybrid systems, parallel and distributed simulation, programming language support for system designs, real-time operating systems, mobile computing, distributed systems, and computer architecture.



Franjo Ivančić received his Diplom degree in Computer Science from the Rheinische Friedrich-Wilhelms University in Bonn, Germany, in 1999 and his M.S.E. Degree in Computer Science from the University of Pennsylvania in 2000. He was a Research Associate at the German National Research Center for Information Technology (GMD) in Sankt Augustin, Germany, from 1997 to 1999. Franjo is currently a Ph.D. candidate of the Department of Computer and Information Science at the University of Pennsylvania. His research interests include software design automation techniques for embedded systems and formal methods for the analysis of hybrid systems.

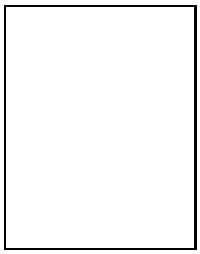


Vijay Kumar received his M.Sc. and Ph.D. in Mechanical Engineering from The Ohio State University in 1985 and 1987 respectively. He has been on the Faculty in the Department of Mechanical Engineering and Applied Mechanics with a secondary appointment in the Department of Computer and Information Science at the University of Pennsylvania since 1987. He is currently a Professor and the Deputy Dean for research in the School of Engineering and Applied Science. He is a member of the American Society of Mechanical Engineers, Institution of Electrical and Electronic Engineers, and Robotics International, Society of Manufacturing Engineers. He has served on the editorial board of the IEEE Transactions on Robotics and Automation, Editorial Board of the Journal of Franklin Institute and the ASME Journal of Mechanical Design. He is the recipient of the 1991 National Science Foundation Presidential Young Investigator award and the 1997 Freudenstein Award for significant accomplishments in mechanisms and robotics. His research interests include robotics, dynamics, control, design, and biomechanics.

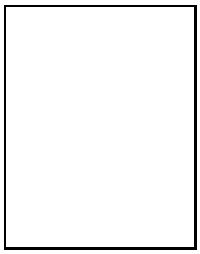


Insup Lee received his B.S. degree in mathematics from the University of North Carolina, Chapel Hill, in 1977, and his Ph.D. degree in computer science from the University of Wisconsin, Madison, in 1983. He is currently Professor in the Department of Computer and Information Science at the University of Pennsylvania, where he has been since 1983. He was CSE Undergraduate Curriculum Chair from September 1994 to August 1997. He has served on numerous program committees, and also (co-)chaired several conferences and workshops. Insup has been on the editorial boards of *IEEE Transactions on Computers*, *Formal Methods in System Design*, and *Journal of Electrical Engineering and Information Science*, and is an IEEE fellow. His research interests include real-time systems, formal methods, and software engineering. He has been developing programming concepts, language constructs, and operating systems for real-time systems. In recent years, he has been developing specification, analysis, and testing techniques based

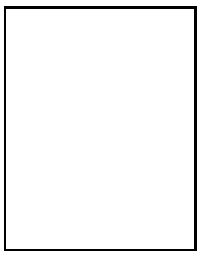
on real-time process algebra (ACSR) and hybrid systems. Furthermore, he has been implementing and evaluating software engineering tools based on formal techniques. He has also developed the MaC framework that can be used to assure the correctness of a running system through monitoring and checking of safety properties.



Pradyumna Mishra received his B.Tech. degree from the Department of Electrical Engineering at the Indian Institute of Technology at Kharagpur in 2000. He is currently a Ph.D. student in the Department of Computer and Information Science at the University of Pennsylvania. His research interests are in the areas of embedded control systems, hybrid systems, distributed robotics, and air traffic management systems.



George J. Pappas received his B.S. degree in Computer and Systems Engineering in 1991, his M.S. degree in Computer and Systems Engineering in 1992, both from the Rensselaer Polytechnic Institute, Troy, NY. In 1994, he was a Graduate Fellow at the Division of Engineering Science of Harvard University. In December 1998, he received his Ph.D degree from the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. He is currently an Assistant Professor and Graduate Group Chair in the Department of Electrical Engineering at the University of Pennsylvania, where he also holds a Secondary Appointment in the Department of Computer and Information Sciences. George is the recipient of the 2002 NSF CAREER award and the 1999 Eliahu Jury Award for Excellence in Systems Research from the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. His research interests include embedded hybrid systems, hierarchical control systems, nonlinear control systems, geometric control theory, flight and air traffic management systems, robotics, and unmanned aerial vehicles.



Oleg Sokolsky received his M.Sc in Computer Science from the St. Petersburg Technical University (Russia) in 1988 and his Ph.D. in Computer Science from SUNY at Stony Brook in 1996. Oleg is currently a Research Assistant Professor at the University of Pennsylvania, where he has occupied research staff positions since 1998. His research interests include formal methods for the analysis of real-time and hybrid systems, model checking and run-time verification.