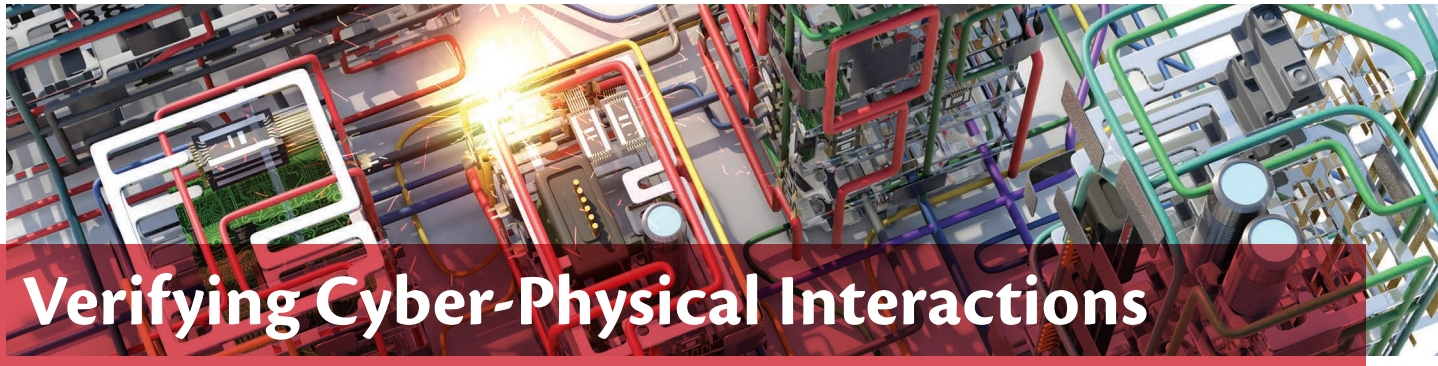

Verifying Cyber-Physical Interactions in Safety-Critical Systems

Sayan Mitra | University of Illinois at Urbana-Champaign
Tichakorn Wongpiromsarn | Singapore-MIT Alliance for Research and Technology
Richard M. Murray | Caltech

Safety-compromising bugs in software-controlled systems are often hard to detect. In a 2007 DARPA Urban Challenge vehicle, such a defect remained hidden during more than 300 miles of test-driving, manifesting for the first time during the competition. With this incident as an example, the authors discuss formalisms and techniques available for safety analysis of cyber-physical systems.

Turning left during the third round of the 2007 DARPA Urban Challenge, Alice—an autonomous Ford Econoline van—dangerously deviated from the computer-generated path and started stuttering in the middle of a busy intersection. Earlier in the competition, Alice had completed two rounds of missions involving on- and off-road driving, parking, merging, and U-turns while obeying traffic rules—all with style and with no human driver. This was a testament to 15 months of programming, debugging, and test-driving by a team of 50 students and researchers from Caltech, Jet Propulsion Laboratory, and Northrop Grumman.

Alice's onboard hardware included 10 cameras; eight laser radars (LADARs); two ordinary radars; an inertial navigation system; two pan-tilt units; 25 CPUs; and actuators for the steering, throttle, brake, transmission, and ignition (see Figure 1). The software included the sensing and the control systems, with 48 individual programs and more than 100 concurrent threads. The control system had modules for making actuation decisions at different spatial and temporal scales on the basis of the processed data streams (see Figure 2a). For instance, the mission planner computed routes for completing high-level missions using a road map, the traffic planner ensured traffic rule conformance on the basis of finite state machines (FSMs), the path planner generated waypoints on the basis of inputs from the upper levels and obstacles, and the controller computed acceleration and steering signals for the vehicle to follow the waypoints. For safety, we "unit tested" each component against reasonable-sounding but informal assumptions about the other components and its own physical environment.

An unforeseen interaction among the control modules and the physical environment, not witnessed during more than 300 miles of autonomous test-driving and hours of extensive simulations, led to this safety violation and Alice's unfortunate disqualification. What happened? We implemented a reactive obstacle avoidance (ROA) subsystem to rapidly decelerate Alice for collision avoidance. Under normal operation, ROA would send a brake command to the controller when Alice got too close to an obstacle or when it deviated too much from the planned path. The controller would then rapidly stop Alice, and the path planner would generate

 Copublished by the IEEE Computer and Reliability Societies

a new path. It turns out that, to protect the steering system, the interface to the physical hardware limits the steering rate at low speeds. If the path planner produces sharply turning paths at low speeds, then Alice can't follow and, thus, deviates. These two effects interacted during the third task, which involved making sharp left turns while merging into traffic. Alice deviated from the path, the ROA activated and slowed it down, and the path planner generated a new path with an even sharper turn to try a successful merge. This cycle continued, taking Alice to the verge of a collision.

It's clear that unforeseen interactions between cyber and physical components can wreak havoc in critical systems from individual control systems in vehicles, factories, and networks of systems in air traffic control and power grids. The Alice incident illustrates that the interaction of software, hardware, and physical components (for example, the ROA and the steering protection) outside the informally assumed environment is often the source of such surprises.

In the context of analyzing Alice, we discuss formalisms and techniques available for safety analysis of cyber-physical systems (CPSs). We also discuss simulation-based approaches, more formal approaches, and the emerging area that attempts to take advantage of both.

Modeling Cyber-Physical Systems

One insight from our postmortem analysis of the incident (presented in "Verification of Periodically Controlled Hybrid Systems: Application to an Autonomous Vehicle"[1]) suggests adoption of open modeling—which forces module designers to precisely pin down all (possibly nondeterministic) behavior of the environment. Then, the component in question is designed and tested in the context of that environment. Analysis based on open models exposes unmodeled or unforeseen interactions. Open models necessarily introduce uncertainties in the form of environmental inputs. This nondeterminism increases the size of the space of possible behaviors that designers must consider during safety analysis, but fortunately, state-of-the-art tools from formal methods and control theory are well poised to tackle such models. Using these open models and analysis techniques, we've been able to diagnose design defects in Alice, air traffic control protocols, and various other control systems.

Simulation Models

Tools such as LabView, Simulink, and Stateflow have made the creation of complex CPS models from simpler building blocks a standard skill for today's engineering graduates. A Simulink-Stateflow model describes a CPS as a collection of interconnected functional blocks—for example, integrators, switches, and filters, and hierarchical state machines. Figure 3 shows a model of part of

Figure 1. Alice, Team Caltech's entry in the 2007 DARPA Urban Challenge. The Ford Econoline van was equipped with 10 cameras; eight laser radars (LADARs); two ordinary radars; an inertial navigation system; two pan-tilt units; 25 CPUs; and actuators for the steering, throttle, brake, transmission, and ignition.

Alice's control system. Such models are used primarily to generate simulation traces. Simulink's simulation engine generates runs of a deterministic model by numerically integrating the model equations over time with a user-specified method. Although simulation-centric tools are indispensable for rapid prototyping, design, and debugging, they're limited in providing safety guarantees. Naively running a finite number of simulations can't eliminate the possibility of safety violations, even for finite-state models that run for a long time. The simulation traces record state information only at discrete points in time. Whatever happens in between requires additional analysis.

Nondeterminism—a key feature for capturing uncertainty and underspecified open models—isn't supported in these tools either. Finally, even for deterministic systems, numerical errors might cause the simulation points to deviate arbitrarily from the actual states visited by the system. Although sophisticated simulation engines vary the sampling period and use different numerical techniques to minimize errors in each simulation step, obtaining global error bounds is tricky, especially in models in which the system dynamics change discontinuously. When Alice's controller changed the steering input to the vehicle based on, for example, a new waypoint, it did so in single discrete steps. Such discontinuous changes are commonplace in CPS models.

Two research directions aim to make simulation tools useful for safety analysis: First, expressive simulation environments, such as Modelica[2] and Ptolemy,[3] have been built from the ground up; the models have precise semantics and, therefore, can be used for rigorous safety analysis. The second approach recognizes the widespread adoption of Simulink and aims to obtain safety guarantees from imperfect simulations. It combines formal static analysis with the simulations' dynamic, inaccurate information.

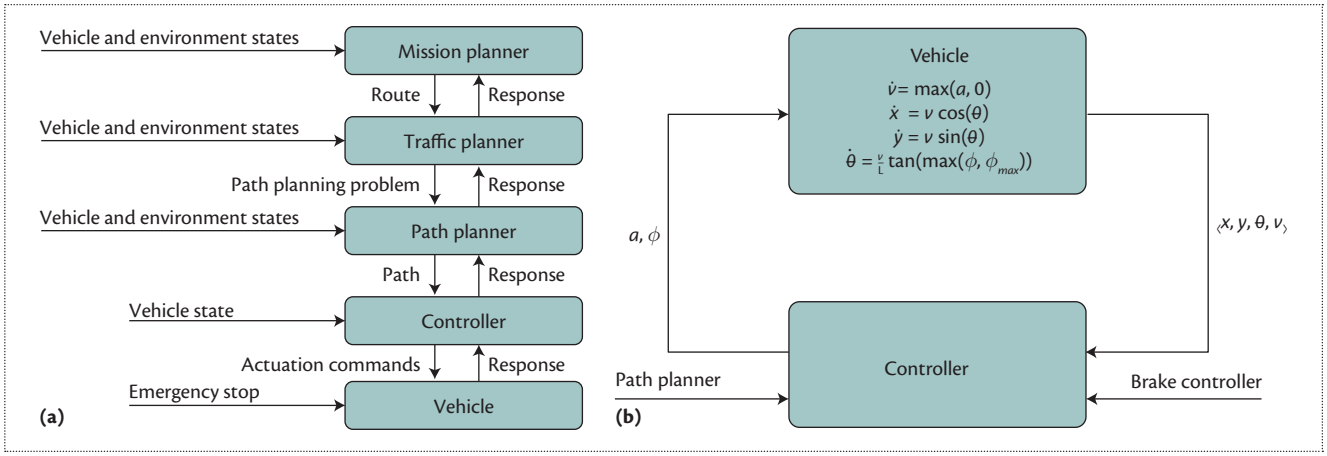

Figure 2. Alice's software. (a) The control protocol stack. (b) Controller-vehicle automata and their interface with path planner and brake controller. Several hierarchical controllers sense the environment and ultimately inform the controller, which computes the acceleration, steering, and brake signals for the vehicle.

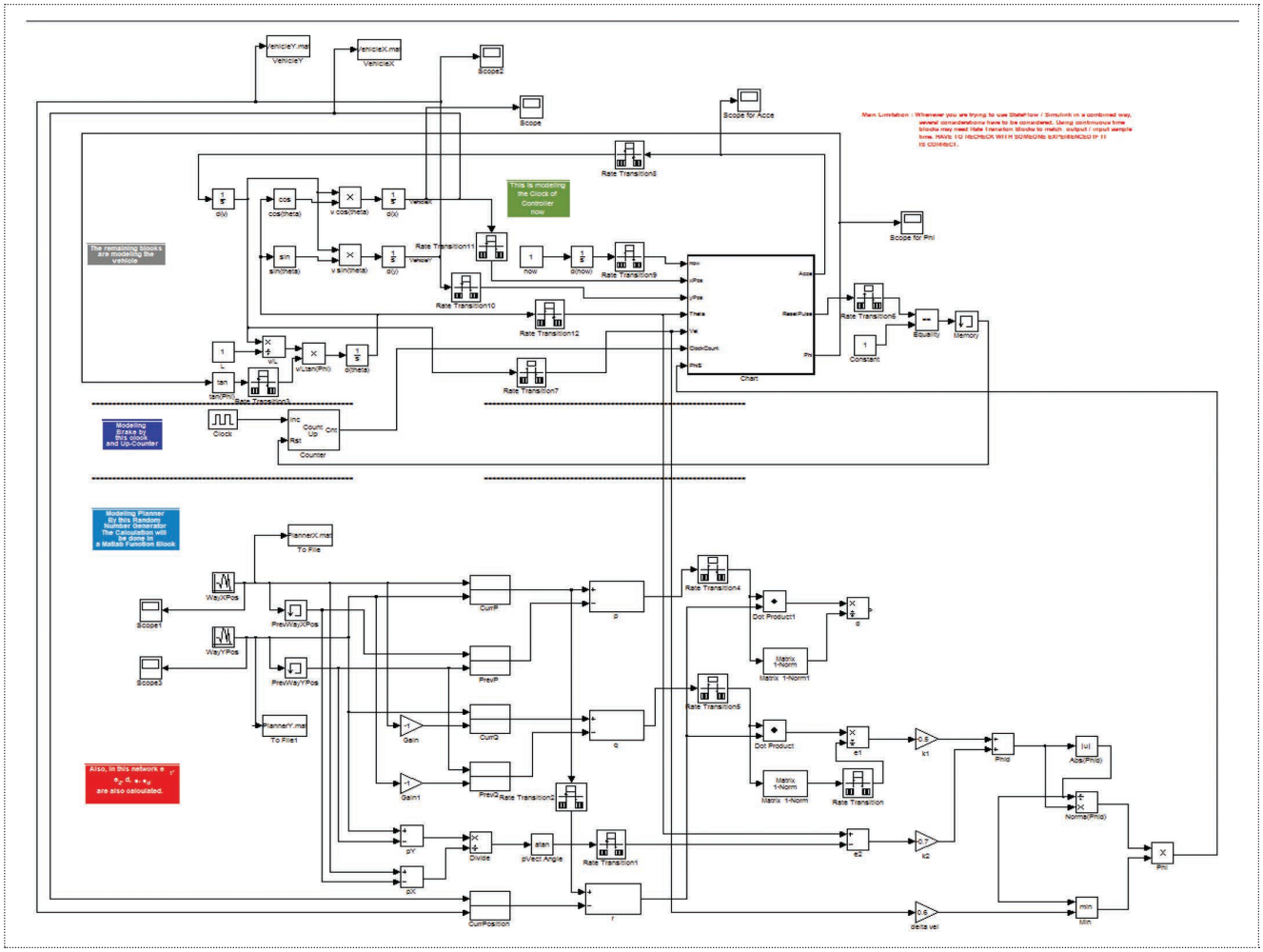

Figure 3. A Simulink/Stateflow model for Alice's control system and its dynamics. Deterministic models such as these can be simulated to generate simulation traces.

Table 1. Partial hybrid I/O automation (HIOA) specifications of the vehicle and controller (details of the function f are omitted).

HIOA model of vehicle	HIOA model of controller
Automaton vehicle(ϕ) **variables** **output** x,y,θ,v:$\mathbb{R}$: = ‹x_0, y_0, θ_0, v_0› **input** α,ϕ: $\mathbb{R}$ **trajectories** **evolve** $\dot{x} = v\cos\theta; \dot{y} = v\sin\theta$ $\dot{\theta} = \frac{v}{L}\tan\max(\phi,\phi_{max})$ **if** $v > 0 \vee \alpha > 0$ **then** $\dot{v} = \alpha$ **else** $\dot{v} = 0$	**Automaton controller**(Δ) **variables** **output** α,ϕ: $\mathbb{R}$: = ‹0,0› **input** x,y,θ,v: $\mathbb{R}$ **internal** clk : $\mathbb{R}$: = 0, $path, brakeReq, \overline{x}, \overline{y}, \overline{\theta}, \overline{v}$ **transitions** **input** plan **pre** true **eff** \\update path **input** brake **pre** true **eff** \\ update brakeReq **internal** update **pre** $clk = \Delta$ **eff** $clk := 0; \overline{x} := x; \overline{y} := y; \overline{\theta} := \theta; \overline{v} := v;$ $\langle\alpha,\phi\rangle$: = $f(\langle$*input and internal vars*$\rangle)$ **trajectories** **evolve** $\dot{clk} = 1$ **stop when** $clk = \Delta$

Formal Models and Properties

Reconciling the differences between the discrete models used for computer hardware and programs and the continuous models used for dynamic systems, the hybrid systems community has developed several frameworks for modeling and analyzing CPSs. The differences between these frameworks have to do with their origins and the analysis types they support. The switched system[4] and the hybrid dynamical system[5] frameworks emerged from control theory and are used to study stability, robustness, controllability, and observability. Computability hasn't been the focus; behaviors are described in terms of abstract mathematical objects. The hybrid I/O automaton (HIOA) framework emerged from the distributed computing literature.[6] It supports open models and has well-developed theories for abstractions and composition. Rajeev Alur and his colleagues' popular hybrid automaton framework was designed to study automatic verification and decidability questions.[7] André Platzer's hybrid dynamic logic supports proof rules for reasoning about systems with many identical components.[8] For this article, these differences are less important than the shared principles. For concreteness, we consider the HIOA framework.

An HIOA is a nondeterministic state machine with well-defined I/O variables and actions. Its state can evolve through discrete transitions, as in an FSM, as well as through continuous trajectories, which are solutions of differential equations. The input variables' behavior and actions can be underspecified, which enables us to write open models. The framework requires the models to be input enabled, which essentially forces the modeler to write down how the automaton must react to any inputs from a given class. We describe simple HIOA models for the vehicle and Alice's controller.

Vehicle. An HIOA state is described in terms of the valuations of a set of state variables. For example, the vehicle automaton's state variables include the position of the vehicle in the plane (x, y), its heading θ, and its speed v (see Table 1). These variables are declared as outputs because we want them to be accessible to the controller. The vehicle also has two input variables: acceleration (a) and steering angle (ϕ). The state variables' trajectories are defined in terms of differential equations (and inclusions) involving the state as well as the input variables. The key point is that for a given class of input signals for a and ϕ—say, piece-wise continuous signals—the differential equations are guaranteed to have solutions that define trajectories for x, y, v, and θ. This captures the notion that the HIOA model—here, the vehicle—should be able to react to all possible behaviors of its environment—namely, the controller. This input-enabling requirement lets us define the semantics of, and therefore analyze, open hybrid models that have underspecified input variables and transitions.

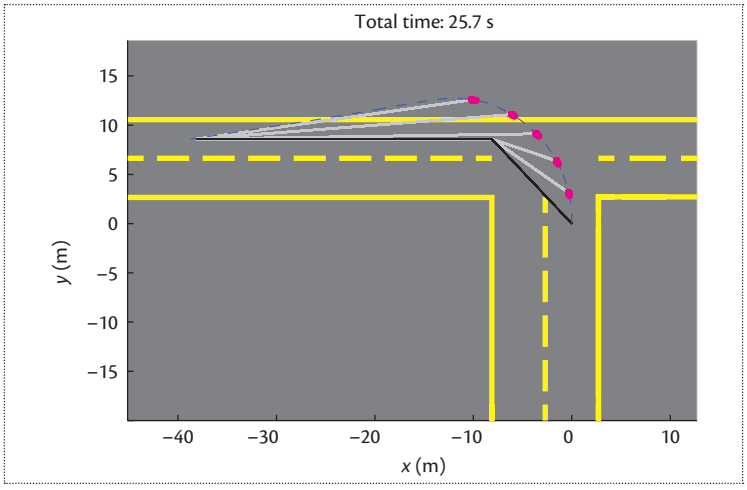

Figure 4. An execution of a vehicle-controller system with input brake transitions. The red dots indicate the occurrence of brake action (triggered by reactive obstacle avoidance), which results in a new path (generated by the path planner) as well as a new controller output. The blue lines indicate the projection of the intervening vehicle trajectories on *x* and *y*.

Controller. Table 1 shows the controller model that periodically reads the vehicle's state and computes the acceleration and steering. The controller also receives interrupts from the path planner and the brake controller. The interrupt information is recorded in state variables, which influence the computation of a and ϕ.

The controller's variables include the last sampled state information $(\overline{x}, \overline{y}, \theta, \overline{v})$, the sequence of waypoints provided by a planner *path*, an internal timer *clk*, information received from the braking unit *brakeReq*, and other internal variables used to compute the controller outputs. The periodic update is modeled by an urgent update action. Because the stopping condition of the trajectories is set to $clk = \Delta$, the *clk* variable is prevented from increasing beyond Δ; the same condition enables the update transition.

When this action occurs, the clock resets to zero, and the outputs a and ϕ are computed using some function of the input and the other state variables. The plan and brake input actions capture the interrupts received from the path planner and the brake controller. Because these actions' occurrences are controlled by the controller's environment, they can potentially occur at any time. Once again, the controller is an open system and has well-defined behaviors in any environment in which it receives the appropriate input actions and variables.

Open models and nondeterminism. The HIOA framework supports modeling of open systems. Uncertainties and underspecification in the external environment and in the automaton itself are modeled as nondeterministic choices. Indeed, there are sources of nondeterminism in a hybrid model that don't exist in purely discrete models. First, the input transitions and trajectories are exogenous to the model, but the model should have well-defined behavior for any input from a reasonable class. Component failures and noise are often modeled as input transitions and variables because the automaton has very little control over them. Second, even without input variables, the differential equations and inequalities might permit multiple solutions over time. For example, a standard way to model a clock *clk* that's drifting with some bounded rate $\pm\rho$ is to write the differential inclusion $(1 - \rho) \leq \dot{clk} \leq (1 + \rho)$. This implies that along any solution of the system, starting from $clk = 0$, the value of *clk* after time t is a point in the interval $[clk_0 + (1 - \rho)\, t,\ clk_0 + (1 + \rho)\, t]$. Finally, at a given state, an automaton might have a choice between several discrete transitions or for some amount of time to elapse. This is exploited for modeling transitions with some uncertainty in the timing behavior. For example, by relaxing the precondition of the update action for the controller to $clk \in [\Delta - \varepsilon, \Delta]$, we could model a slightly less stringent time-triggered controller that updates its output once every $[\Delta - \varepsilon, \Delta]$ time. In addition, as in the case of discrete models, there might be nondeterministic choices among multiple start states and multiple enabled actions in the initial state.

Semantics: Executions, Reachability, Invariants, Safety, and Robustness

We introduced hybrid models through examples of Alice's subsystems; here, we introduce some concepts related to the semantics of such models.

The semantics of a state machine-based modeling framework such as HIOA are defined in terms of states and executions. The system's state is defined by valuations of all the variables in the model. Predicates or constraints on the variables can be interpreted as sets of states. For example, the semantic interpretation of the predicate $Deviation(\varepsilon) = dist(x, y, path) \leq \varepsilon$ is the set of states of the system such that the position is within $\in$ distance (measured by some metric *dist*) of the path.

An HIOA execution is an alternating sequence of its transitions and trajectories. Figure 4 shows the plot of an execution with the backdrop of a left turn. The red dots indicate the occurrence of brake action (triggered by the ROA), which then results in a new path (generated by the path planner) as well as a new controller output. The blue lines indicate the projection of the intervening vehicle trajectories on *x* and *y*.

Because a hybrid automaton is nondeterministic, it has a set of executions that captures all its behaviors. A state is said to be reachable if there exists an execution that terminates at that state. The set of all such states, $Reach_A$, is called the *reach set*. A safety property S is specified by predicates on variables. We could assert that

Table 2. Example correctness requirements.

Correctness requirement type	Examples
Safety or invariance: Reach set of *A* is contained in some safe set *S*, or *A* never goes outside *S*.	The vehicle controller system never deviates more than 1 m from the straight line joining the successive waypoints, for any planner and brake controller behavior.
	No two aircrafts in an air traffic management system ever come within *d* distance.
Progress: For sets of states *I* and *F*, every execution starting from *I* reaches *F* within bounded time.	Starting near a waypoint, the vehicle reaches the next waypoint within *T* time, provided there are no incessant brake requests. *T* depends on current state, the waypoints, and the issued brake requests.
	Starting from safe states, every aircraft eventually gets clearance to land.

Deviation(2 m) is a safety requirement, meaning that the system's reach set is contained in the set *Deviation(2 m)*.

An overapproximation of $Reach_A$ is called an *invariant set*. Invariants can provide useful semantic information about a system. For instance, for programs, states outside an invariant are never reached. This leads to a useful approach for proving safety: find or compute an invariant set that's disjoint from the given unsafe set (complement of the safety property).

Reachability and invariance have their bounded-time counterparts. A state that's reachable within a time horizon of *T* is *T-reachable*, and so on. We will say *A* is ε*-safe* with respect to a set *S* within time *T* if all states within distance ε from some state reachable within time *T* are within *S*. *A* is *robustly safe* if it's ε-safe for some ε. Often, the verification approaches are designed to establish robust safety.

Abstractions

A key benefit of formal modeling is that it lets us precisely define abstractions. Semantically, a hybrid automaton *B* is an abstraction of another automaton *A* if every execution of *A* is also an execution of *B*. (In general, only the visible parts of the executions have to match up, which provides more flexibility in defining what's important.) An abstraction *B* overapproximates the behaviors *A*, and therefore, if B is constructed to be more tractable than *A*, then by proving its safety, we can infer *A*'s safety.

The HIOA framework provides rules to establish abstraction relationships between open models. However, even though this definition of abstraction is useful for reasoning in terms of the containment of executions, it doesn't lead to algorithms for computing the abstraction *B*. Instead, we compute some sort of *simulation relation* that encodes the abstraction. A forward simulation relation from *A* to *B* relates the variables of *A* and *B* such that for every transition and trajectory of *A*, there exists a corresponding transition (or trajectory) of *B* that reserves the relation. Various different notions of forward and backward simulation relations have been developed for hybrid modeling frameworks. The importance of abstractions and simulation relations in automated safety analysis will become evident later.

Safety Verification Tools and Techniques

Here, we discuss three approaches available for rigorous correctness analysis or verification of safety-critical systems like Alice. Two classes of properties capture most of the common correctness requirements in systems, namely, safety and progress (see Table 2). Here, we focus on the extensively studied problem of safety verification. (We refer the interested reader to "Lyapunov Abstractions for Inevitability of Hybrid Systems" for some recent results on verification of progress and stability.[9]) The tools and algorithms underlying these verification approaches have also been applied to synthesize controller code that's correct by construction.[10,11]

Automatic Reach Set Overapproximations

The automatic procedure for proving that a hybrid model *A* is safe with respect to some unsafe set *U* involves computing the (unbounded) reach set $Reach_A$ of *A* and checking that $Reach_A$ and *U* are disjoint. If $Reach_A$ is computable, this procedure proves safety, and whenever the sets aren't disjoint, the procedure finds bugs by giving executions that lead to *U*.

Classes of decidable hybrid automata have been identified for which $Reach_A$ can be computed exactly. These include hybrid automata with only clocks, automata with variables that only evolve at constant bounded rates and are reset whenever the rates change, and so-called o-minimal and stormed hybrid models. Other decidable classes are obtained when the number of continuous variables is small. The core idea in all these results is a bisimulation argument that constructs an FSM that can exactly mimic the transitions and trajectories of the hybrid automaton *A* being analyzed. We can then compute the reach set of this bisimilar FSM by exploring its control graph to obtain $Reach_A$.

A distinguished history of software tools—UPPAAL, HyTech, d/dt, Checkmate, PHAVer, and the more recent SpaceEx—embody these reachability algorithms

and have been applied to successfully analyze the safety of a variety of CPS models.[12,13] In each step along the way, new insights about the data structures used for representing the partially computed reach sets (for example, polyhedra, rectangles, and ellipsoids) have played an important role in improving the algorithms' efficiency.

However, for general CPSs, decidable hybrid automata models are few and far between. For most models with more than a small number of continuous variables and linear or nonlinear dynamics, computing $Reach_A$ exactly is undecidable. Even for a decidable automaton, obtaining scalable exact reachability algorithms is a big challenge. For instance, the vehicle-controller model we described would require stretching the capabilities of current reachability tools. To address this, we relaxed the reachability problem to compute over $Reach_A$ approximations through abstractions.

The hybridization scheme in "Accurate Hybridization of Nonlinear Systems" proposes a specific type of abstraction in which the complicated nonlinear dynamics of A are approximated by piece-wise linear or piecewise rectangular dynamics in B.[14] The state space of A is partitioned into regions, and within each region, the complex nonlinear vector field is overapproximated by a set-valued rectangular (or linear) vector field.

Hybridization is one of several different methods for constructing property-agnostic abstractions. The partitioning and overapproximation we described don't rely on the unsafe set U. In contrast, counterexample guided abstraction refinement (CEGAR) algorithms, which have been successful in software verification, construct abstractions for proving or disproving a specific safety property.

CEGAR starts with a coarse abstraction B_0 of A; in each iteration, $Reach_{B_i}$ is computed and checked for safety. If B_i is safe, then A is safe, and the algorithm terminates. Otherwise, B_i is unsafe and a counterexample a is produced (provided the $Reach_{B_i}$ instances can be computed). If a corresponds to a valid execution of A, then we can immediately infer that A is unsafe. This is called the *validation step*. Otherwise, a is a spurious counterexample arising from the overapproximation in B_i and is used to obtain a new refined abstraction B_{i+1}, and the procedure continues. If a CEGAR algorithm terminates with "safe" or "unsafe," the answer is correct, but it's guaranteed to terminate only if

- the abstract automata are from a decidable class,
- the validation step is computable, and
- the refinement step eliminates enough spurious counterexamples to make progress toward a decision.

The existing CEGAR algorithms for restricted classes of hybrid models show promise of improved scalability; however, termination guarantees are likely to be achievable only under additional robustness assumptions. For instance, termination might be guaranteed if the automaton A is not only safe with respect to U but also robustly safe. When A is safe but not robustly safe, the algorithm might not terminate. Development of property-directed abstraction-refinement schemes for verification of nonlinear hybrid models remains an open problem.

Finding Inductive Invariants with Human Guidance

An alternative to automatically computing overapproximations of $Reach_A$ or invariants of the given system model is to find *inductive invariants*. An invariant I is inductive for the system A if

- all start states of A satisfy I, and
- starting from a state that satisfies I, A continues to satisfy I following a transition or a trajectory.

Given a predicate I, checking whether it's an inductive invariant is typically easier than computing the reach set. For discrete transitions, this involves symbolically executing the model for all possible transitions that are enabled from I. For the continuous trajectory, this can be achieved by checking that the vector field defining the evolution of A "points inward" at the boundary of I. This so-called *subtangential condition* can be checked symbolically without necessarily solving the differential equations.

This is the approach we used to establish the key safety properties of a corrected model of Alice.[1] Often, the initial guess I must be strengthened by introducing constraints on the state variables. For example, to establish that the deviation of the vehicle-controller model from the planned path is upper-bounded by some ε_{max}, we must also introduce bounds on the disorientation of the vehicle with respect to the path's direction. The vehicle's inertia, the limits on its maneuverability, and the controller periodicity require that for the deviation to be bounded, the vehicle can't be highly disoriented. Specifically, we defined a collection of predicates I_k (for each natural number k) that incorporates the following constraints:

- The deviation from the path is bounded by a constant ε_k, for a constant $\varepsilon_k \leq \varepsilon_{max}$.
- The disorientation is small enough such that the steering angle computed by the controller (as a function of the deviation and disorientation) is in the range $[-\phi_k, \phi_k]$ for a constant ϕ_k that doesn't exceed the physical limit of the vehicle steering capacity.
- The speed is bounded by a constant v_{max}.

Figure 5 shows the collection of predicates I_k projected onto the deviation-disorientation plane. Provided

that the brakes aren't triggered too frequently, the path's turns aren't too sharp compared to its length, and the controller's execution speed isn't too slow with respect to v_{max}, each I_k is in fact an inductive invariant of the vehicle-controller system. This establishes the bounded deviation (safety) property. The analysis makes the notion of too frequent, too sharp, and too slow precise in terms of the vehicle and controller parameters. Checking inductive invariance of I_k using the subtangential condition is a routine exercise once the correct set of assumptions is in place. As an added bonus, the invariants also aid the progress analysis. During each control period, as the vehicle makes progress toward the next waypoint, we show that under the above assumptions, it moves from I_k to I_{k+1} that's contained in I_k. That is, in executing a long segment, the vehicle converges to a small deviation and disorientation with respect to the path, so the instruction for executing a subsequent sharp turn doesn't make the deviation and disorientation grow too much.

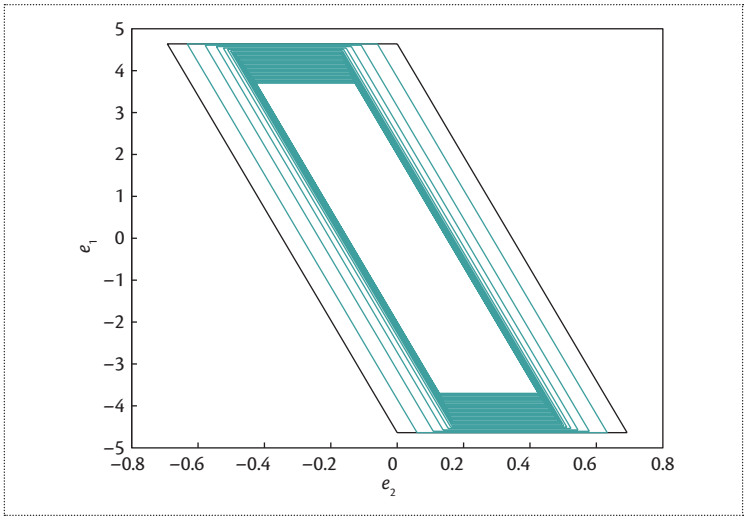

Figure 5. The invariant sets for different values of deviation and disorientation for Alice's control software. During each control period, as the vehicle makes progress toward the next waypoint, it moves from an outer polytope I_k to an inner polytope I_{k+1}.

Toward Automatic Search for Inductive Invariants

The earlier approach is applicable to a general class of open CPS models. When successful, it provides useful information about the system's behavior apart from just establishing safety. For instance, by checking inductive invariants for the open system—the vehicle and the controller—we derived restrictions on its environment—the brake controller and the path planner—that were sufficient for safety. The main limitation is the first guess. Finding useful inductive invariants requires creativity and insight about the system's behavior.

To address this issue, Sriram Sankaranarayanan and his colleagues proposed several methods for finding inductive invariants of hybrid models automatically.[15] In these approaches, the insights are encoded in the form of invariant templates. These templates define the invariants' possible shape and must be instantiated to find the actual invariants from that shape class. For example, the search for inductive invariants is seeded by a family of, in this case, linear template constraints $F(\overline{c},\overline{x}) = \sum_i c_i x_i \leq d$, where $\overline{c}$ and d are parameters to be synthesized. Then, the boundary of the invariant set is given by the set of points $\overline{x}$ where $\sum_i c_i x_i = d$ and the subtangential condition can be encoded as the constraint

$$\forall \overline{x}[\sum_i c_i x_i = d \Rightarrow \frac{d(\sum_i c_i x_i)}{dt} < 0] .$$

In this linear case, the Lagrange relaxation of the resulting linear programs can be solved to automatically find linear invariants and, in some cases, even the strongest linear invariants. Several interesting open problems arise in identifying subclasses of nonlinear models and more general templates for which the resulting set of constraints can be solved effectively to yield a richer variety of inductive invariants.

Reaping Benefits of Simulations for Verification

The final verification approach we discuss takes us back to simulation. As we mentioned, simulation models are used widely, and (possibly inaccurate) simulation traces are inexpensive to generate in a pragmatic sense. How can we overcome the semantic imprecisions and ambiguities of these simulation models and use a finite but possibly large number of simulations traces to obtain formal safety guarantees? The "proofs from tests" or "verification from simulations" paradigm has produced success stories in hardware and software verification and is now being developed for CPS.[16–19] The key idea is to compute, from an individual simulation trace (or a test) that starts from a single start state x_0, a tube that contains all the system's executions starting from an open ball around x_0. Because this ball has a nonzero radius, it's possible to compute tubes that overapproximate all executions from a set of initial states from a finite number of simulation traces and, therefore, to verify safety from all initial states.

This approach's success for CPS hinges on the ability to compute these tubes for hybrid models from simulation traces that record only discrete snapshots. The gaps in the trace must be filled to construct the tubes that are guaranteed to contain the executions. Furthermore, as we mentioned, the recorded traces from a state

x_0 might have arbitrarily large errors compared to the actual states that are visited in the execution starting from x_0. For deterministic models, control theoretic properties of the models, such as stability, contractiveness, and continuity, can be used to compute these tubes.[18–20] For example, if the right-hand side of the differential equations are Lipchitz continuous or if the system executions are asymptotically stable, then the tube containing all executions starting from a ball around x_0 can be computed in a straightforward manner. Checking a dynamical system's asymptotic stability isn't generally an easy problem; however, it's reasonable to expect that the simulation models are annotated with certificates that ensure these control theoretic properties (for example, Lyapunov functions, Lipchitz constants, and contraction metrics) hold. As long as the simulation model meets the annotations, the procedure's soundness and completeness can be guaranteed for bounded time analysis, even if the model's exact formal semantics aren't known.

Consider a deterministic hybrid automaton A with a set of initial states Q_0 and an unsafe region U. Using the above approach to compute tubes, we can check whether A is robustly safe with respect to U up to a time bound T by simulating A from finitely many initial states. The stability properties ensure that for every $\varepsilon > 0$, there's a computable $\delta > 0$ such that for any two executions starting within a δ-ball remain within an ε-ball up to time T. If U is an open set of unsafe states and Q_0 a bounded set of initial states, then under certain mild assumptions about the continuity of A's executions, the robust safety problem can be decided by simulating A from finitely many initial states.

This approach has been explored in several papers and implemented in the Breach[18] and C2E2[19,20] tools. It offers a significant advantage over the reachability-based approaches we discussed. First, those methods can theoretically work only for systems whose continuous flows are described by polynomials and even then are unlikely to scale to large systems because of the need to reason about nonlinear arithmetic. Second, because simulation engines can work with systems with complicated nonlinear dynamics, and the subsequent analysis of simulation traces doesn't concern itself with the actual dynamics, they could potentially handle systems that can't be handled by today's model checkers.

Although this line of research is still in its early stages, the preliminary experimental results are very promising and handily outperform reachability-based tools in natural examples. Development of a simulation-based verification framework for nondeterministic models, possibly including actual controller code, remains an open problem. In particular, for distributed systems with message delays, clock skews, and scheduling uncertainties, handling nondeterminism poses major challenges in generating traces, identifying useful annotations, and developing effective safety verification algorithms.

Our experiences with the safety analysis of Alice and several other real-world CPS models strongly suggest that component-level verification alone isn't sufficient to guarantee safety. Open and formal modeling and verification should be applied to identify dangerous and unpredictable component interactions. The current state-of-verification tools provide several options for analyzing different kinds of models with different levels of investment. For systems with linear dynamics and a dozen or so continuous variables, completely automatic safety analysis is becoming feasible on a modest verification budget. For more general models, mechanical verification guided by human inputs (for example, in the form of inductive invariant templates) is possible with a larger budget. Finally, widely adopted simulation models can be leveraged for obtaining bounded time safety guarantees for high-dimensional nonlinear models. Although social, legal, and economic barriers remain for widespread adoption, we believe that computer-aided safety analysis tools are ready to enter engineers' inventory. ■

Acknowledgments

This research was partially supported by US Air Force Office of Scientific Research (AFOSR) through the Multidisciplinary University Research Initiative (MURI) program and a research grant from the National Science Foundation (1016791).

References

1. T. Wongpiromsarn et al., "Verification of Periodically Controlled Hybrid Systems: Application to an Autonomous Vehicle," *ACM Trans. Embedded Computing Systems*, vol. 11, no. S2, 2012, art. 53.
2. S. Mattsson, M. Otter, and H. Elmqvist, "Modelica Hybrid Modeling and Efficient Simulation," *Proc. 38th IEEE Conf. Decision and Control*, IEEE, 1999, pp. 3502–3507.
3. E.A. Lee et al., *Overview of the Ptolemy Project*, tech. report UCB/ERL M01/11, Dept. Electrical Eng. and Computer Science, Univ. California, Berkeley, 2001.
4. D. Liberzon, "Switching in Systems and Control," *Systems and Control: Foundations and Applications*, Birkhauser, 2003.
5. R. Goebel, R.G. Sanfelice, and A.R. Teel, *Hybrid Dynamical Systems: Modeling, Stability, and Robustness*, Princeton Univ. Press, 2012.
6. D.K. Kaynar et al., *The Theory of Timed I/O Automata. Synthesis Lectures on Computer Science*, Morgan Claypool, 2005.

7. R. Alur et al., "The Algorithmic Analysis of Hybrid Systems," *Theoretical Computer Science*, vol. 138, no. 1, 1995, pp. 3–34.
8. A. Platzer, "Differential Dynamic Logic for Hybrid Systems," *J. Automated Reasoning*, vol. 41, no. 2, 2008, pp. 143–189.
9. P.S. Duggirala and S. Mitra, "Lyapunov Abstractions for Inevitability of Hybrid Systems," *Hybrid Systems: Computation and Control*, ACM, 2012, pp. 115–124.
10. C. Belta et al., "Symbolic Planning and Control of Robot Motion," *IEEE Robotics & Automation Magazine*, vol. 14, no. 1, 2007, pp. 61–70.
11. H. Kress-Gazit, G.E. Fainekos, and G.J. Pappas, "Temporal-Logic-Based Reactive Mission and Motion Planning," *IEEE Trans. Robotics*, vol. 25, no. 6, 2009, pp. 1370–1381.
12. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi, "Hytech: A Model Checker for Hybrid Systems," *Computer Aided Verification* (CAV 97), LNCS 1254, Springer, 1997, pp. 460–483.
13. G. Frehse, "Phaver: Algorithmic Verification of Hybrid Systems Past Hytech," *Hybrid Systems: Computation and Control*, LNCS 3414, Springer, 2005, pp. 258–273.
14. T. Dang, O. Maler, and R. Testylier, "Accurate Hybridization of Nonlinear Systems," *Proc. 13th ACM Int'l Conf. Hybrid Systems: Computation and Control* (HSCC 10), ACM, 2010, pp. 11–20.
15. S. Sankaranarayanan, H.B. Sipma, and Z. Manna, "Constructing Invariants for Hybrid Systems," *Formal Methods in System Design*, vol. 32, no. 1, 2008, pp. 25–55.
16. T. Nghiem et al., "Monte-Carlo Techniques for Falsification of Temporal Properties of Non-Linear Hybrid Systems, *Proc. 13th ACM Int'l Conf. Hybrid Systems: Computation and Control* (HSCC 10), ACM, 2010, pp. 211–220.
17. P. Zuliani, A. Platzer, and E.M. Clarke, "Bayesian Statistical Model Checking with Application to Simulink/Stateflow Verification," *Proc. 13th ACM Int'l Conf. Hybrid Systems: Computation and Control* (HSCC 10), ACM, 2010, pp. 243–252.
18. A. Donzé, "Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems," *Proc. 22nd Int'l Conf. Computer Aided Verification* (CAV 10), Springer, 2010, pp. 167–170.
19. Z. Huang and S. Mitra, "Computing Bounded Reach Sets from Sampled Simulation Traces," *Proc. 15th ACM Int'l Conf. Hybrid Systems: Computation and Control* (HSCC 12), ACM, 2012, pp. 291–294.
20. P. Duggirala, S. Mitra and M. Viswanathan, *Verification of Annotated Models from Executions*, tech. report UILU-ENG-13-2204 (CRHC-13-04), Coordinated Science Laboratory, Univ. Illinois at Urbana-Champaign, June 2013; https://publish.illinois.edu/c2e2-tool.

Sayan Mitra is an assistant professor of electrical and computer engineering at the University of Illinois at Urbana-Champaign. His research interests include formal methods, distributed systems, and cyber-physical systems. Mitra received a PhD in electrical engineering and computer science from MIT. He received the National Science Foundation's CAREER award in 2011, the AFOSR Young Investigator Research Program Award in 2012, and several best paper awards. Contact him at mitras@illinois.edu.

Tichakorn Wongpiromsarn is a postdoctoral associate at the Singapore-MIT Alliance for Research and Technology. Her research interests span hybrid systems, distributed control systems, formal methods, transportation networks, and situational reasoning and decision making in complex, dynamic, and uncertain environments. Wongpiromsarn received a PhD in mechanical engineering from the California Institute of Technology. Contact her at nok@smart.mit.edu.

Richard M. Murray is the Thomas E. and Doris Everhart Professor of Control & Dynamical Systems and Bioengineering at Caltech. His research is in the application of feedback and control to networked systems, with applications in biology and autonomy. Murray received a PhD in electrical engineering and computer sciences from the University of California, Berkeley. Contact him at murray@cds.caltech.edu.