 The original paper was published in *American Control Conference (ACC), 2012*. IEEE, 2012. http://dx.doi.org/10.1109/ACC.2012.6315388

# Using Theorem Provers to Guarantee Closed-Loop System Properties

Nikos Aréchiga[†], Sarah M. Loos[‡], André Platzer[‡], Bruce H. Krogh[†]

[†]Department of Electrical and Computer Engineering {narechig|krogh}@ece.cmu.edu
[‡]Department of Computer Science {sloos|aplatzer}@cs.cmu.edu
Carnegie Mellon University, Pittsburgh, PA 15213-3890

***Abstract*— This paper presents a new approach for leveraging the power of theorem provers for formal verification to provide sufficient conditions that can be checked on embedded control designs. Theorem provers are often most efficient when using generic models that abstract away many of the controller details, but with these abstract models very general conditions can be verified under which desirable properties such as safety can be guaranteed for the closed-loop system. We propose an approach in which these sufficient conditions are static conditions that can be checked for the specific controller design, without having to include the dynamics of the plant. We demonstrate this approach using the KeYmaera theorem prover for differential dynamic logic for two examples: an intelligent cruise controller and a cooperative intersection collision avoidance system (CICAS) for left-turn assist. In each case, safety of the closed-loop system proved using KeYmaera provides static sufficient conditions that are checked for the controller design.**

## I. Introduction

Formal methods for verification apply logical analysis to well defined mathematical models to determine whether or not mathematically precise specifications are satisfied. The power of formal methods lies in the precision of the result: correctness is guaranteed if a formal method concludes the specification is satisfied, at least in so far as the model correctly represents the system being analyzed. Tools that implement formal methods are used regularly in the design and analysis of digital circuits [8], and there are a growing number of successful applications of formal methods in other domains [17], [16]. The application of formal methods to verify the correctness of automotive control system designs, and control systems in general, has been limited, however, for several reasons. Most tools used for control system development do not use the same modeling formalisms that are the foundation of the development of formal methods in computer science. In particular, models for control design typically include differential equations or other representations of continuous dynamics that do not arise in strictly digital applications. Although the controller itself is usually a digital system, specifications for control systems are for the closed-loop system behavior, so it is the properties of the combined digital and continuous dynamic system that need to be evaluated, not just the input-output behavior of the digital component.

Research in hybrid dynamic systems focuses on precisely these types of systems that arise in control, that is, systems with both discrete and continuous dynamics [7]. Tools for formal analysis of hybrid systems [5], [6], [13], [14] cannot deal directly with the complex models used for automotive control system design, however. Consequently, to analyze closed-loop system behavior formally, simplified models need to be constructed for which the verification problem is tractable. If the simplified models are *abstractions*, meaning the set of behaviors represented by the simplified model include all possible behaviors of the complex model, verifying the correctness of the simplified model implies the correctness of the complex model. Thus, the common proposal for applying formal methods to control systems (and even digital systems) is to first construct an abstraction of the system design and then formally verify the correctness of this abstraction. Some automatic abstraction processes have been proposed [4], [14].

This paper proposes a different approach to leveraging the power of formal methods to guarantee the correctness of control system designs. Rather than constructing an abstraction of a particular design, we propose that an abstraction be constructed that captures the possible behaviors of the closed-loop system for an entire class of controllers. Theorem proving is used to verify critical properties, such as safety, for the closed-loop system, for any controller in this general abstraction of possible controllers. The proof of correctness takes into account the continuous dynamics of the plant being controlled. To verify the correctness of a particular controller, it is necessary to show only that the controller is in the class of controllers represented by the abstraction used for formal verification. This second step, verifying that the controller is represented by the verification model, is considerably less complex than verifying the closed-loop system is correct because the plant dynamics do not need to be included. The results from formal verification become a set of conditions on the input-output behavior of the controller that need to be checked. Provided these conditions are satisfied, the designer is free to focus on other aspects of the design, such as specifications for the transient response of the system.

The approach described above is similar in some respects to the concept of *refinement* in software development, e.g., [1], where formal methods are used to verify the correctness of the requirements specifications and then the implementation is constructed by"refining" the abstract model that has been verified. The difference in our proposal for control system verification is that the controller is a refinement of only part of the closed-loop system that is verified formally. The continuous dynamics that introduce much of the difficulty in formal verification are not included in the part of the

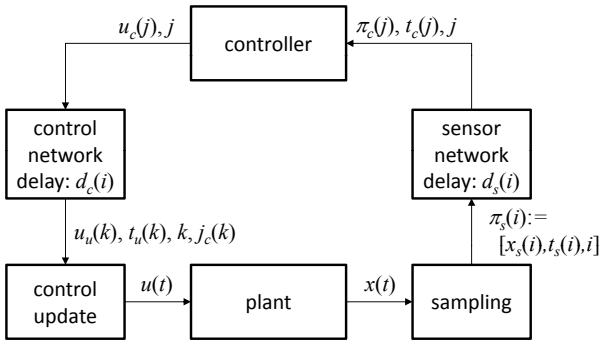


Fig. 1. A control system with general network communication (see text for signal definitions).

system that is "refined". Refinement has been proposed in other contexts or for special systems in the embedded domain [2], [15], [3]. We make the refinement ideas constructive by using theorem proving to obtain guarantees about the system behavior.

The following section describes the proposed approach more formally. Section III explains differential dynamic logic and the hybrid system theorem prover KeYmaera, with which properties of very general closed-loop computer-controlled dynamic systems can be verified. We use KeYmaera proofs to provide conditions on the input-output mapping realized by the digital controller. We illustrate the proposed approach for safety verification for two automotive control examples in Sections IV and V, respectively: an intelligent cruise control system and a cooperative intersection collision avoidance system. The concluding section summarizes the contributions of this paper and discusses directions for further research.

## II. General Approach

We consider the general control system scenario illustrated in Fig. 1. The plant is described by a continuous-time state equation

$$\dot{x} = f(x, u), \tag{1}$$

where $x$ is the plant state and $u$ is the control input. The control input is generated by a sampled-data controller that is connected to the plant via a sensor network that delivers the sampled values from the sensors to the control computer and a control network that delivers the control values from the computer to the plant.

The signals shown in Fig. 1 are defined as follows. The plant state is sampled at times $0 < t_s(1) < t_s(2) < \cdots$ and the sampled values of the state at these sample times are denoted by $x_s(i) = x(t_s(i))$, $i = 1, 2, \ldots$ .[1] At the $i^{th}$ sampling time, the packet $\pi_s(i) = [x_s(i), t_s(i), i]$ is sent through the sensor network to the controller. The controller receives these data packets at times $t_c(1) < t_c(2) < \cdots$, but because of the sensor network delays $d_s(i)$, which can be different for each packet, the order in which the packets arrive may not correspond to the order of the sampling sequence. To model this possibility, we let $i_s(j)$ denote the index of the sample time at which the $j^{th}$ received sample was taken. Thus, the value of the $j^{th}$ packet received by the controller is given by $\pi_c(j) = [x_s(i_s(j)), t_s(i_s(j)), i_s(j)]$ and $t_c(j) = t_s(i_s(j)) + d_s(i_s(j))$.

[1]For simplicity of notation, the full state is sampled in the model. The functions that determine the sensor values that are actually available to the controller can be incorporated into the controller model.

The controller is modeled by the general discrete-time state equations

$$\begin{aligned} z(j+1) &= g(z(j), \pi_c(j), t_c(j), j) \\ u_c(j) &= h(z(j), \pi_c(j), t_c(j), j), \end{aligned}$$

where $z$ is the internal controller state and $u_c$ is the controller output. The controller state is updated at each control instant $t_c(j)$ when a new packet is received over the network. We assume the initial controller state $z(0)$ is given, and the control update begins with the first packet received ($j = 1$). The information available to the controller includes the time stamp and index for the sampled state value in $\pi_c(j)$, so that strategies for dealing with networked communications are included in this general model of the controller.

The controller sends the control value $u_c(j)$ and its index over the control network to the plant, where the value of the control input $u(t)$ is updated. As with the sensor network, the control network introduces a delay $d_c(j)$. Therefore, the control values received at the plant to update the control input, denoted $u_u(k)$, are not necessarily received in the same order as the control values that were computed and sent by the controller. Control values and their indices $j_c(k)$ are received at the plant at times $t_u(1) < t_u(2) < \cdots$, and $t_u(k) = t_c(j_c(k)) + d_c(j_c(k))$.

The control update at the plant is given for k=1,... by

$$u(t) = r(u_u(k), t_u(k), k, j_c(k)) \quad \text{for } t_u(k) \le t < t_u(k+1). \tag{2}$$

We assume an initial control value $u(t) = u_0$ is specified for the time $0 \le t < t_u(1)$ before the first control update is received from the controller. The resulting control input is piecewise continuous, being updated at the update times $t_u(k)$, and the value of the update at each instant is a general function of the received updated control value, its time stamp and index, and the original index assigned by the controller. Note that any information can be included in the control values $u_c(j)$, including the time stamp for the data upon which it is based.

Verifying the correctness and safety of a control system of the type described above is very difficult, particularly for specifications that apply to the closed-loop behavior. In the standard approach to verification, the problem is made tractable by creating a simplifying abstraction of the controller, and then verifying the properties of the abstraction. It is often very difficult and time consuming to construct a verifiable abstraction of a complex control system. Indeed, this is perhaps the primary reason formal verification and theorem proving has not yet become a common tool for control system engineering.

In this paper, we propose a different approach which starts with the construction of a verifiable model. Rather than build an abstraction of a particular controller design, we begin with

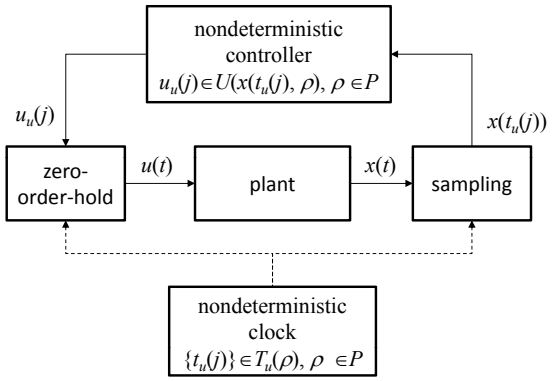


Fig. 2. A nondeterministic sampled-data control system for verification using a theorem prover.

a nondeterministic model of a feedback control system that represents all possible behaviors of the closed-loop system for a very general class of controllers. The model used by the theorem prover to verify specifications for the closed-loop system is illustrated in Fig. 2. In this model, the control input is updated by the zero-order-hold to be value $u_u(j)$ at each update time $t_u(j)$. The control update values $u_u(j)$ are selected from a set of possible control values $U_u(x(t_u(j)), \rho)$, where $\rho$ is a vector of parameter values selected from a set $P$. Moreover, the model represents all possible behaviors for any sequence of update times chosen from some general class of sequences $T_u(\rho)$. Although this model does not contain all details of the control system in Fig. 1 explicitly, it is a type of closed-loop system for which properties can be verified using a theorem prover. Moreover, this model can represent a very general class of possible controllers, where only the features that facilitate the proof of the desired closed-loop properties are included in the model.

Given the proof that the closed-loop system satisfies the desired specification for any controller for which $\rho \in P$, $T_u(\rho)$, and $u_u(j) \in U(x(t_u(j)), \rho)$, it suffices to show that a particular control design satisfies this condition on the control input as a function of the value of the state at each instant when the control input is updated at the plant. If this can be done, the closed-loop property for the system is guaranteed, *without having to evaluate the closed-loop system behavior using the particular controller*.

Suppose, for example, that there are no network delays and the controller is a constant state-feedback gain of the form $u_u(j) = Kx(t_u(j))$ and that the sampling sequence corresponds to a parameter value $p = p' \in P$. In this case it would be sufficient to show that $Kx \in U(x, p')$ for all $x$. When the controller has an internal state or there are network delays, demonstration that a particular control design satisfies the condition given by $U(x(t_u(j)), \rho)$ is more difficult because the value of the control input at time $t_u(j)$ is not a direct function of $x(t_u(j))$, the value of the state at time $t_u(j)$. In general, the control input $u(t_u(j))$ depends on some past values of the controller state and sampled plant state. That is, $u(t_u(j)) = F(z_c', x_s')$, where $z_c'$, $x_s'$ are past values of the controller and plant states, respectively, and the times when these state values were determined depend on the possible network delays. Moreover, the controller state value depends on the possible histories of the plant state that have driven the controller state equation.

To take advantage of the proof given by the theorem prover in this more general situation, it is necessary to determine a range of possible values for $z_c'$ and $x_s'$ for any possible value of $x(t_u(j))$. This will determine the range of possible values that the control input may have for a given value of $x(t_u(j))$. If this range can be determined, i.e., if one can find a set $\hat{U}(x(t_u(j)), \rho)$ such that $F(z_c', x_s') \in \hat{U}(x(t_u(j)), \rho)$ for all values of $z_c'$ and $x_s'$ that are possible knowing $x(t_u(j))$, it is then guaranteed $u(t_u(j)) \in \hat{U}(x(t_u(j)), \rho)$. Then it suffices to show that $\hat{U}(x(t_u(j)), \rho) \subseteq U(x(t_u(j)), \rho)$ to guarantee the closed-loop system satisfies the property established by the theorem prover. If this can be done, the desired property of the control system design is guaranteed without having to analyze the closed-loop system.

Before illustrating the above approach for two automotive control system examples in Sections IV and V, we present the details of a theorem prover that can be used to establish closed-loop properties of sampled-data control systems in the following section.

## III. Theorem Proving Using Differential Dynamic Logic

We specify system models with controller plant loops and their properties in *differential dynamic logic* (dL) [11], [12], which has been implemented in the theorem prover KeYmaera. In dL, we describe controller plant loops (or, more generally, hybrid systems) in a program notation. Such *hybrid programs* involve both the discrete controller actions and the continuous dynamics. Part of the hybrid program captures the controller, where a programming language is a natural notation. Another part of the hybrid program directly allows one to state the plant dynamics. We can think of hybrid programs as conventional programs with extra differential equations inside that model the continuous system dynamics. Hybrid programs can also be understood as the control code that is generated in model-based design, but enriched with explicit differential equations to retain the continuous plant. The logic dL and the prover KeYmaera are described in detail in [12].

To illustrate, we consider a simple example of a car at position $p$ with velocity $v$ driving down a straight road. The Simple Car Control (SCC) sets the acceleration of the vehicle based on sensor values of state variables $p$ and $v$. The controller may choose either to accelerate ($A$) or brake ($-B < 0$) until the car passes a start braking point $sb$, at which time the car must apply the brakes until it comes to a stop. The controller does not have continuous access to the sensor values for state variables $p$ and $v$, but receives updates periodically, therefore the car may overshoot $sb$ slightly before braking. In the remainder of this section we use dL to formally model a general class of Simple Car Controllers (SCC) as a hybrid program which satisfies the above description.

**Model 0** Simplistic Car Controller (*SCC*) in dℒ

$$SCC \equiv (ctrl; dyn)^* \quad (3)$$

$$ctrl \equiv (a := -B) \cup (?\textbf{Safe};\ a := A) \quad (4)$$

$$\textbf{Safe} \equiv p \leq sb \quad (5)$$

$$dyn \equiv p' = v, v' = a\ \&\ v \geq 0 \quad (6)$$

The hybrid program SCC is shown in Model 0. The program shows step by step how the model executes. Line (3) says that, first, the controller *ctrl* executes and then (after the sequential composition ;), the continuous dynamics *dyn* evolves for a certain period of time. In Model 0, the execution of the controller *ctrl* completes instantly without any time passing. The continuous dynamics *dyn*, instead, continues for some time (≥0). This process (first do *ctrl*, then do *dyn*) repeats, as indicated by the repetition operator * at the end of (3), because the controller *ctrl* will again have a chance to react to situation changes after the continuous evolution went on for a certain period of time. This repetition $(ctrl; dyn)^*$ models the sampled-data controller-plant feedback loop. It is part of the flexibility of hybrid programs that we are not forced to specify exactly how long the continuous evolution continues before the controller reacts again. With that, we can easily model several variations in the controller design and phenomena like jitter in a single model.

The controller in (4) has the option to brake ($a := -B$) or to accelerate ($a := A$) and may choose between both options nondeterministically (written ∪). The option $a := A$, however, is only available when the system state successfully passes the test ?**Safe** that it is safe to accelerate. In the particular design in (5), this test is simply to check whether the current system state satisfies $p \leq sb$Note here, that the nondeterminism in (4) leaves the controller both options when the current system state passes the test ?**Safe**, but only leaves the option $a := -B$ when it does not pass test ?**Safe**. This nondeterminism is actually beneficial for the verification, because, if we can verify that Model 0 is safe, then any implementation with a particular strategy for arbitrating the nondeterministic choices would also be safe. For example, a controller that would only brake when it is not safe to accelerate. Or a controller that additionally optimizes for fuel consumption. The operations in *ctrl* are just assignments, choices, and tests, and, thus, consume no (noticeable) time, so *ctrl* indeed terminates instantly, as expected from a controller in a hybrid system. The *ctrl* operations, thus, determine at any instant of time when they run the set of controls $U(x(t_u(j)), \rho)$ at that state. This gives a set, because of the nondeterminisms in *ctrl* and the choices of the parameters $\rho$ (including *sb*). Since the controllers are not assumed to run continuously, they correspond to the feedback of data sampled at certain times, here simply $t_u(j)$.

The continuous plant dynamics *dyn* for Model 0 in (6) is the continuous state equation for movement of $p$ according to velocity $v$ and acceleration $a$, but it is restricted to (written &) the evolution domain region $v \geq 0$. This evolution domain region makes sure that the car never starts to drive backwards even when braking ($a := -B$).

More generally, for polynomial or rational terms $\theta$, *hybrid programs* allow arbitrary other combinations of assignments ($x := \theta$), nondeterministic assignments ($x := *$), differential equation systems ($x' = \theta\ \&\ H$) with evolution domain constraint $H$, arithmetic constraints $H$ as tests ($?H$), nondeterministic choices (∪), sequential compositions (;), and nondeterministic repetitions (*). See [11], [12] for details.

The advantage of representing the model formally in dℒ is that we may guarantee safety properties which the controller will never violate (or, if the controller is flawed, it returns a situation which violates the safety properties which is helpful for debugging). Additionally, because we used nondeterminism to model a general class of controllers rather than any specific implementation, all specific refinements of the non-deterministic model will inherit the safety guarantees for free.

For instance, suppose in the simple example described above the road ends at point $s$ and the car must come to a stop before that point. We want to prove that, from any initial state satisfying some conditions *start*, there is no circumstance where the *SCC* model will allow the car to drive past the end of the road ($p < s$). We represent this by the following dℒ formula:

$$start \rightarrow [SCC](p < s) \quad (7)$$

The formula *start* is called a precondition that we assume to hold before the system *SCC* runs. And the formula $p < s$ is the postcondition that we conjecture to hold at all time after executing *SCC*. For the simplistic Model 0, it turns out that dℒ formula (7) is not true, because the model does not specify a bound on how long the plant *dyn* could evolve before the controller *ctrl* again has a chance to react to situation changes. The safety statement (7) is not true for arbitrary delays. A more realistic model would, thus, include a bound $t \leq \varepsilon$ on the time $t' = 1$ how long the plant evolution in (6) can continue without interruption by the controller. We illustrate these phenomena in detail in the sequel.

## IV. Example 1: Intelligent Cruise Control

Intelligent Cruise Control (ICC) adjusts the acceleration of a car based on the velocity and position of the car directly ahead which has been sampled by sensors or from vehicle to vehicle (V2V) communication. In this scenario, the follower car is the one with the ICC and is called the *subject vehicle* (SV). The lead car is called the *primary other vehicle* (POV). The ICC must control the acceleration of the SV in a way that will not cause a collision, no matter when or how hard the POV brakes. This means that a crash must be avoided even when the POV slams on the brakes and the SV does not notice that the POV is slowing until it receives sensor data after a maximum delay. We assume that sensor update delay is bounded by $\varepsilon$. In this example, we show that a general class of controllers for ICC is safe. We then show that a specific PID controller is a refinement of the general ICC, and therefore inherits the safety guarantee.

An illustration of the ICC scenario is shown in Fig. 3.

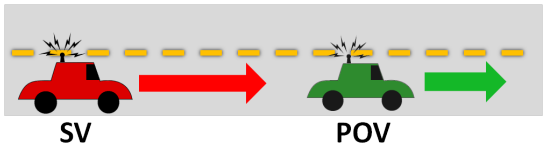


Fig. 3. Diagram of the intelligent cruise control scenario

**Model 1** Intelligent Cruise Control (*ICC*) in dℒ

$$ICC \equiv (ctrl; dyn)^* \tag{8}$$
$$ctrl \equiv \text{POV}_{ctrl} \parallel \text{SV}_{ctrl}; \tag{9}$$
$$\text{POV}_{ctrl} \equiv (a_{\text{POV}} := *;\ ?(-B \le a_{\text{POV}} \le A)) \tag{10}$$
$$\text{SV}_{ctrl} \equiv (a_{\text{SV}} := *;\ ?(-B \le a_{\text{SV}} \le -b)) \tag{11}$$
$$\cup\ (?\mathbf{Safe}_\varepsilon;\ a_{\text{SV}} := *;\ ?(-B \le a_{\text{SV}} \le A)) \tag{12}$$
$$\cup\ (?(v_{\text{SV}} = 0);\ a_{\text{SV}} := 0) \tag{13}$$
$$\mathbf{Safe}_\varepsilon \equiv p_{\text{SV}} + \frac{v_{\text{SV}}^2}{2b} + \left(\frac{A}{b} + 1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon v_{\text{SV}}\right) < p_{\text{POV}} + \frac{v_{\text{POV}}^2}{2B} \tag{14}$$
$$dyn \equiv (t := 0;\ t' = 1, \tag{15}$$
$$p'_{\text{SV}} = v_{\text{SV}},\ v'_{\text{SV}} = a_{\text{SV}}, p'_{\text{POV}} = v_{\text{POV}},\ v'_{\text{POV}} = a_{\text{POV}} \tag{16}$$
$$\&\ (v_{\text{SV}} \ge 0\ \wedge\ v_{\text{POV}} \ge 0\ \wedge\ t \le \varepsilon)) \tag{17}$$

### A. dℒ Model and Proof of Safety

Model 1 shows a generic class of ICCs written as a hybrid program in dℒ. The discrete control of the POV is modeled in line (10) as $\text{POV}_{ctrl}$. Since we do not assume that the POV is using any specific ICC, its acceleration is set non-deterministically to any value within the physical limits of the vehicle, thereby capturing the behavior of any sampled-time controller.

The model of the discrete control of the ICC for the SV is represented as $\text{SV}_{ctrl}$ in Model 1. Line (11) allows the car to brake at any time, and assumes that the brakes are at least as strong as $-b$ and no stronger than $-B$. Line (12) says that, so long as the car is within the safety envelope defined by $\mathbf{Safe}_\varepsilon$ in line (14), the acceleration of SV is set non-deterministically within the physical limits of the vehicle. Finally, line (13) simply says that if the car is stopped, it may choose to remain stopped. Because acceleration is set non-deterministically, when safety of Model 1 is proved, it must hold for all possible values of acceleration. Therefore, if a specific controller can be shown to respect the safety envelope (i.e. always applying the brakes when $\mathbf{Safe}_\varepsilon$ does not hold), then it inherits the safety verification of Model 1.

The continuous dynamics of the cars are represented by the differential equations in *dyn*. We assume here that the cars are traveling down a straight road, that they are a single point, and that sensors have no delay and are accurate. Some of these assumptions will be not be required in the following sections. The evolution domain in line (17) additionally requires that the cars may brake to a stop but they will never reverse (i.e. velocity is always non-negative). It also assumes that the SV receives fresh data about the position and velocity of the POV within a maximum of $\varepsilon$ time.

To prove that the ICC is safe, we show that at all times the SV is behind the POV ($p_{\text{SV}} < p_{\text{POV}}$), given the cars begin in a controllable state ($\mathbf{Safe}_0 \wedge (p_{\text{SV}} < p_{\text{POV}})$). This is expressed in dℒ by the following formula, which has been proved by the automated theorem prover KeYmaera:

$$\mathbf{Safe}_0 \wedge (p_{\text{SV}} < p_{\text{POV}}) \rightarrow [ICC](p_{\text{SV}} < p_{\text{POV}})$$

This proof has 924 nodes, requires 656 user interactions, and proves in 329 seconds. For a detailed description of this model and its proof see [9].

### B. PID Controller Design

Now that we have system bounds that have been proved safe, it is possible to design a more realistic controller to the specifications required by Model 1. We design a control system model of an ICC which models a sampled-time PID controller deterministically setting the acceleration of the SV. We leverage the proof of safety for a general class of ICCs from Sect. IV-A to show that safety is still guaranteed under this specific PID instance of an ICC.

The PID controller has a desired set point, $d_{set}$, which represents the distance between the SV and the POV that the controller tries to maintain. The controller only operates while the vehicles are traveling within a set minimum and maximum velocity, $v_{min}$ and $v_{max}$. When the velocity bounds are not met, we assume another controller with the same safety guarantees takes control. The PID gains are denoted by $K_p, K_i, K_d$. We also bound the integral term from accumulating above $z_{max}$ and below $z_{min}$. The controller then has the following, standard form:

$$h(x_S(t), z_S(t)) = \begin{cases} a_{PID} & \text{if } -B \le a_{PID} \le A \\ A & \text{if } a_{PID} > A \\ -B & \text{if } a_{PID} < -B \end{cases}$$

where, for some $z \in [z_{min}, z_{max}]$:

$$a_{PID} = K_p((p_{\text{POV}} - p_{\text{SV}}) - d_{set}) + K_i \cdot z + K_d(v_{\text{POV}} - v_{\text{SV}})$$

In order for any specific, deterministic model to inherit the proof of safety for Model 1, we must prove it is a refinement of Model 1. That is, when a specific implementation of the ICC makes a control decision, it falls within the range of control decisions permissible in Model 1 under the same conditions. For a sampled-time PID controller, the acceleration of the SV is set based on the position and velocity of the two vehicles, the gains, and the desired distance between the two vehicles, as shown in $a_{PID}$. If the acceleration or braking returned by the PID formula exceeds the physical limits of the vehicle, the controller caps it at that limit.

To prove that a sampled-time PID controller is a refinement of Model 1, it suffices to check that when the safety boundary is violated (i.e. $\mathbf{Safe}_\varepsilon$ is false), the PID controller applies emergency braking, as shown in Model 1, line (11). This is sufficient because, in the alternative case where $\mathbf{Safe}_\varepsilon$ is true, the proof of safety holds for any choice of

acceleration within the physical limits of the vehicle. The PID controller satisfies this requirement by design.

We implemented a simple proof of refinement in KeYmaera which tests algebraically whether, given the parameters of a fully specified PID controller, the formula for acceleration would yield only values below $-b$ whenever $\mathbf{Safe}_\varepsilon$ is false. The proof is split into two files which together have 114 nodes, require 46 user interactions, and prove in 1.8 seconds. If a PID controller does not pass this test, then it has not been proved to qualify as a refinement of Model 1.

With a simple modification, we were able to change the test for refinement into a search for a refinement which, for a given subset of parameters of a PID controller, outputs constraints on the remaining parameters. For example, given the PID gains, $v_{min}$, $v_{max}$, $z_{min}$, $z_{max}$, $A$, $b$, and $B$, the refinement search would output a value of $d_{set}$ which would preserve safety. Because testing refinement only requires algebraic manipulation, there are a number of programming and mathematical tools, besides KeYmaera, which could have implemented these refinement tests.

### C. Design for Delay Tolerance

In Sect. IV-B we showed that a non-deterministic, abstract model of a system, which is far removed from a real world implementation, can be refined into a specific PID controller without sacrificing any guarantees on safety. If we also want the controller to apply to other phenomena not modeled in the original system, we just need to ensure that it still checks against the static constraint. Therefore, even though the original model was of a relatively simplistic controller, because of the flexibility gained by its non-determinism, it becomes a powerful tool in verifying safety for increasingly complex controllers.

For instance, Model 1 makes a simplifying assumption that messages and sensor data are passed instantly between vehicles, so that control choices are made based on current values $p(t)$ and $v(t)$. Unfortunately this would be impossible to implement in a real system. We can easily create a new model which inserts a time delay between the sensors and the controller so that control decisions are based on old values $p(t-d)$ and $v(t-d)$, where $d$ is the delay between the sensors and the controller. This new model also requires a modified formula for the safety boundary. This safety boundary is more conservative than the original boundary, $\mathbf{Safe}_\varepsilon$, since there is less accurate information about the current position and velocity of the POV.

We could reprove safety completely for the updated model; however, repeating a proof of safety for this more complex system would be at least as difficult as the full proof of safety for Model 1. We can avoid this work by proving that the model with delays is still just a refinement of our original model. The computational complexity of proving this refinement is a fraction of the complexity of a full proof of safety. Our proof of refinement is completed in two steps: 1) A proof that the value of the out-of-date, dynamic, sensor data is always within specific bounds, and 2) A proof that for all values within those bounds, the new safety boundary implies the original $\mathbf{Safe}_\varepsilon$.

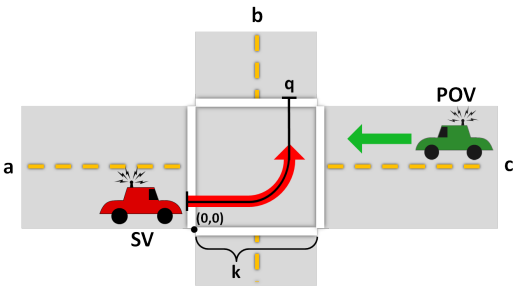


Fig. 4. Diagram of the CICAS-SLTA scenario

This refinement differs from the refinement to a PID controller, since it maintains the non-determinism and overall structure of Model 1 and the resulting model is still written in $\mathsf{d}\mathcal{L}$. This means that it represents a general class of controllers, and could itself be refined further, for example, into a PID controller, using a similar process as described in Sect. IV-B. By piling one refinement onto another in this way, we can model increasingly complex systems and more importantly, we can verify safety properties which were previously only tractable for simple models.

## V. Example 2: A Cooperative Intersection Collision Avoidance System

The Signalized Left Turn Assist (SLTA) is a Cooperative Intersection Collision Avoidance System for intersections with permissive left turns. A permissive left turn is one without a specific left-turn traffic light phase. Left-turn accidents account for 27.3 % [10] of intersection related crashes in the US. An SLTA system allows a traffic engineer to address safety issues at an intersection without adding a separate turning phase, which would restrict traffic flow.

A diagram of the SLTA scenario is shown in Fig.4. The subject vehicle (SV) is stopped in front of an intersection on leg $a$ and intends to turn left onto leg $b$, while the primary other vehicle (POV) is approaching from leg $c$, possibly putting the two automobiles on a collision path.

Both vehicles are equipped with sensors to measure their position and velocity, as well as communication capabilities to relay their measurements to a computer that is monitoring activity at the intersection.

The researchers from the California PATH Project [10] have built a prototype SLTA system that estimates the arrival time of the SV at the intersection assuming it maintains a roughly constant velocity, but they did not formally verify their system. In this example, we create a model of the SLTA scenario in $\mathsf{d}\mathcal{L}$ and use the theorem prover KeYmaera to derive static constraints for the design of a provably safe controller.

### A. $\mathsf{d}\mathcal{L}$ Model

Model 2 shows a general class of controllers modeled in $\mathsf{d}\mathcal{L}$.

**Model 2** Signalized Left Turn Assist (SLTA) in dℒ

$$SLTA \equiv (ctrl; dyn)^* \tag{18}$$

$$ctrl \equiv \text{POV}_{ctrl} \parallel \text{SV}_{ctrl} \tag{19}$$

$$\text{POV}_{ctrl} \equiv (a_{\text{POV}} := *; ?(-A < a_{\text{POV}} < B); T_{\text{POV}} := \frac{k - p_{\text{POV}}}{-v_{max}} \tag{20}$$

$$\text{SV}_{ctrl} \equiv (?(p_{\text{SV}} = 0 \wedge v_{\text{SV}} = 0); \tag{21}$$

$$a_{\text{SV}} := *;\ ?(a_{\text{SV}} \geq a);\ ?(a_{\text{SV}} \leq A); \tag{22}$$

$$T_{\text{SV}} := \left(-v_{\text{SV}} + \sqrt{v_{\text{SV}}^2 - 2a_{\text{SV}}(p_{\text{SV}} - q)}\right)/a_{\text{SV}}; \tag{23}$$

$$?(T_{\text{POV}} > T_{\text{SV}})) \tag{24}$$

$$\cup\ (?(v_{\text{SV}} = 0);\ ?(p_{\text{SV}} = 0);\ a_{\text{SV}} := 0) \tag{25}$$

$$\cup\ (?(p_{\text{SV}} \geq 0);\ ?(p_{\text{SV}} \leq q);\ a_{\text{SV}} := *; \tag{26}$$

$$?(a_{\text{SV}} > a);\ ?(a_{\text{SV}} < A)) \tag{27}$$

$$\cup\ (?(p_{\text{SV}} \geq q);\ a_{\text{SV}} := *;\ ?(-A < a_{\text{SV}} < B)) \tag{28}$$

$$dyn \equiv (t := 0;\ t' = 1,$$

$$p'_{\text{SV}} = v_{\text{SV}},\ v'_{\text{SV}} = a_{\text{SV}}, p'_{\text{POV}} = v_{\text{POV}},\ v'_{\text{POV}} = a_{\text{POV}} \tag{29}$$

$$\&\ v_{\text{SV}} \geq 0 \wedge v_{\text{POV}} \leq 0 \wedge v_{\text{POV}} < -v_{max} \wedge t \leq \epsilon) \tag{30}$$

Line (20) describes the controller of the POV. Since a controller for the SV cannot control the POV, the POV assumes an arbitrary acceleration within a range that represents physical limits of the car. $A$ and $B$ are the maximum possible acceleration and braking, respectively. The acceleration of the POV must be larger than $-A$ and greater than $B$ because it is traveling in a negative direction. Lines (21) through (28) describe the behavior envelope for the SV controller. When the SV is stopped in front of the intersection, the SV may always remain stopped, or it may choose to attempt a turn. If it chooses to attempt a turn, it computes the time $T_{\text{POV}}$ at which the POV would enter the intersection and computes the time the SV would require to execute a turn along a trajectory of length $q$ using a minimal acceleration $a$. If these computations show that it is safe to turn, the controller may choose an acceleration between the lower and upper bounds $a$ and $A$, respectively. If the SV is already inside the intersection, it is required to continue turning so that it does not obstruct the intersection.

When the SV controller computes its estimate of the time of arrival of the POV, it assumes that the POV instantaneously changes its velocity to some upper bound $v_{max}$ and rushes to the intersection. The POV enters the intersection when its position is less than $k$, and the SV will enter the intersection when its position is greater than 0. The SV leaves the intersection when its position is greater than or equal to $q$. The domain of evolution (30) enforces that neither of the two cars may drive in reverse, and that the POV must drive at a velocity lower than the upper bound $v_{max}$.

The model assumes that sensors are perfectly accurate and that information about the POV is received by the SV instantaneously.

To prove that this class of controllers is safe, we use KeYmaera to prove that Model 2 satisfies the property

$$(p_{\text{POV}} < k) \rightarrow (p_{SV} < 0 \vee p_{SV} \geq q),$$

that is, whenever the POV enters the intersection, the SV has either not yet entered it or has already left it, given that the cars begin in a state where the SV is stopped outside of the intersection. We use $\mathbf{SV}_{stopped}$ to represent the condition that the SV is initially stopped outside of the intersection. The safety of Model 2 is expressed in dℒ as shown in the equation below.

$$\mathbf{SV}_{stopped} \rightarrow [SLTA]((p_{\text{POV}} < k) \rightarrow (p_{SV} \leq 0 \vee p_{SV} \geq q))$$

This safety property has been proved using KeYmaera.

As a result of this analysis, we have derived that a controller will be safe if it receives instantaneous information about the POV and requires the SV to remain stopped unless the safety requirement in line (24) holds, i.e. $(k - p_{\text{POV}})/ - v_{max} > \sqrt{2q/a}$, in which case, SV may accelerate with a minimum acceleration of $a$. Additionally, the SV must continue to accelerate if it has already entered the intersection. Then we can say that a controller is safe if it respects

$$h(x_S(t)) \begin{cases} 0 & \text{if } (k - p_{\text{POV}})/ - v_{max} \leq \sqrt{2q/a} \\ u \in (a, A) & \text{if} p_{\text{SV}} > 0 \\ u \in (a, A) \cup \{0\} & \text{otherwise} \end{cases} \tag{31}$$

where $a$ and $A$ are system parameters.

Note that this is a static condition on the input-output relation of the controller.

### *B. Controller Design*

We can now use the static input-output derived from Model 2 to design a safe controller. To ensure that our design is safe, it is sufficient to show that its behaviors are a refinement of the behaviors exhibited by the general class of controllers of the model above. Particularly, we want to design a controller that does not allow the vehicle to enter the intersection and mandates an acceleration of zero when (31) does, and forces a vehicle that has already entered the intersection to continue at a minimum acceleration.

To design this controller, we assume the vehicles have onboard equipment to measure their positions and velocities, and transmit this data to a central station at the infrastructure. When the SV arrives at the intersection, it establishes a communication link with the infrastructure to find out if there is a POV coming, and if so to obtain position and velocity readings for it.

Model 2 assumes instantaneous information, but in order to build a more realistic controller, we would like to allow the information about the POV to arrive with a known, bounded delay. We still assume that the SV knows its own state immediately. To extend our safety result to a situation with a maximum delay of $d$, we need to show that the controller remains safe even when it is acting on stale information, provided it knows an upper bound on just how stale that

information is. Let $h_C(x_S(t-d))$ represent the action of the controller we are trying to design, and $p_{POV}(t_0)$ and $p_{POV}(t_0-d)$ be the position of the POV at time $t_0$ and the reading the controller receives at $t_0$, which is actually the position of the car at $t_0 - d$. Then our controller will remain safe if:

$$\frac{k - p_{POV}}{-v_{max}} \leq \sqrt{2q/a} \rightarrow h_C(x_S(t-d)) = 0 \tag{32}$$

This reduces to the following constraint, which we will need to ensure that our controller respects (31)

$$\frac{k - p_{POV}(t_0)}{-vmax} - d \rightarrow h_C(x_S(t-d)) = 0 \tag{33}$$

Now suppose we want to use a different approach for deciding the safety of the intersection. Instead of assuming that the POV instantaneously switches to the speed limit, assume that it engages some maximal acceleration until it reaches the speed limit and then remains at that constant velocity. Additionally, suppose the controller allows some margin of time between the arrival of the POV at the intersection and the departure of the SV. Then the expression for the decision rule of our controller is

$$h_C(x_S(t)) = \begin{cases} a & \text{if } T_{POV}(x_S(t)) > T_{SV}(x_S(t)) + t_{PE} \\ a & \text{if } p_{sv} > 0 \\ 0 & \text{otherwise} \end{cases}$$

where

$$T_{POV}(x_S(t)) = \frac{k - p_{POV}}{-v_{max}} + \frac{v_{max}}{2A}$$

$$T_{SV}(x_S(t)) = \sqrt{\frac{2q}{a}}.$$

The remaining task is to select $t_{PE}$ such that (33) is satisfied. It is clear that the smallest value that accomplishes this is $v_{max}/(2A) + d$.

As a result, we show that the use of a theorem prover to analyze the behavioral envelope of a general class of controllers produces a static constraint that is sufficient for safety. This static constraint can be easily used to check the safety of a more specific controller design.

## VI. Conclusions

This paper presents an approach for guaranteeing properties of closed-loop control systems using a theorem prover applied to a representation of a general class of controllers. The result from the theorem prover provides a static condition that must be satisfied by the control design. This static condition characterizes a set of allowed control values as a function of the state of the plant at the instants when the control input value is updated. The theorem prover KeYmaera is presented as a tool for proving properties of closed-loop systems with continuous dynamic plants and general sampled-data controllers. We illustrate with two automotive control examples how the static conditions established by KeYmaera can be applied to controllers with internal states and network delays. While the examples in this paper are relatively simple, more complex systems which may include disturbance or uncertainty can also be modeled in $d\mathcal{L}$ [11].

Directions for further work include developing general methods for mapping the details of controller designs in to the types of static state-dependent mappings used in the theorem proving model, as well as extending this work to liveness constraints. We are also investigating parameterizations in the abstraction of the controller used in the theorem prover to facilitate the process of evaluating control design alternatives. A long-term objective of this work is to create tools that will make it possible for practicing engineers to apply theorem proving to problems of practical interest.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1035800, NSF CAREER Award CNS-1054246, and NSF EXPEDITION CNS-0926181. The second author is supported by a DOE Computational Science Graduate Fellowship.

## References

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, volume 1165 of *LNCS*. Springer, 1996.

[3] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In M. D. D. Benedetto and A. L. Sangiovanni-Vincentelli, editors, *HSCC*, volume 2034 of *LNCS*, pages 33–48. Springer, 2001.

[4] E. M. Clarke, A. Fehnker, Z. Han, B. H. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. J. Found. Comput. Sci.*, 14(4):583–604, 2003.

[5] G. Frehse. PHAVer: algorithmic verification of hybrid systems past HyTech. *STTT*, 10(3):263–279, 2008.

[6] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011.

[7] T. A. Henzinger. The theory of hybrid automata. In *LICS*, pages 278–292, Los Alamitos, 1996. IEEE Computer Society.

[8] C. Kern and M. R. Greenstreet. Formal verification in hardware design: a survey. *ACM Trans. Design Autom. Electr. Syst.*, 4(2):123–193, 1999.

[9] S. M. Loos, A. Platzer, and L. Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In M. Butler and W. Schulte, editors, *FM*, LNCS. Springer, 2011.

[10] J. Misener and et al. Cooperative intersection collision avoidance system (CICAS): Signalized left turn assist and traffic signal adaptation. Technical report, University of California, Berkeley, 2010.

[11] A. Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, 2008.

[12] A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010.

[13] A. Platzer and J.-D. Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008.

[14] S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *Trans. on Embedded Computing Sys.*, 6(1):8, 2007.

[15] J. Romberg and C. Grimm. Refinement of hybrid systems. In *Languages for System Specifications*, pages 315–330, 2004.

[16] F. Wang. Formal verification of timed systems: a survey and perspective. *Proc. IEEE*, 92(8):1283 – 1305, aug. 2004.

[17] M. H. Zaki, S. Tahar, and G. Bois. Formal verification of analog and mixed signal designs: A survey. *Microelectronics Journal*, 39(12):1395–1404, 2008.