Formal Methods for Robotic System Control Software

Yanni Kouskoulas, André Platzer, and Peter Kazanzides

Creating software for controlling robotic machinery has unique challenges. This article describes a formal method called differential-dynamic logic (d$\mathcal{L}$) that can help produce zero-defect algorithms for robotic systems. We take the reader through an example of applying d$\mathcal{L}$ to a version of a control algorithm used in an experimental surgical robot. This case study is a simplified variant of an existing control algorithm. It shows how this tool can be useful and illustrates general principles that readers can use when applying this technique to other systems. We describe how to model a control algorithm for the robot and are able to prove that it safely enforces tool movement for a single boundary. Our proof provides a guarantee of the control algorithm's safe behavior for all possible inputs and is far more comprehensive than what is possible by using testing alone.

INTRODUCTION

Motivation

In many instances, programs are concerned only with processing or manipulating data and displaying them to a user, who becomes the agent that ends up taking physical action. However, in some instances, we create software to control other analog devices or machinery directly. We call these hybrid systems because they exhibit a mixture of discrete behavior from the software and continuous behavior from the analog physics of the device being controlled.

From an engineering perspective, creating zero-defect control software for hybrid systems has unique challenges. On one hand, for an engineering system consisting only of analog devices, we can use continuous mathematics to model it and prove that its design satisfies our requirements. On the other hand, for software that only processes data, we can begin to apply the formal methods (i.e., program logics) that we have developed to prove properties about software-only sys-

tems. However, a hybrid system contains the interface between these two domains and requires new logics.

Formal Methods

To guarantee that the control algorithm predictably influences the machinery with which it interacts, we need to develop hybrid logics that are tailored to include both models of discrete programs as well as the continuous equations that govern the analog components.

Engineers can write code to model a hybrid system and then run these models with particular inputs to see how the system behaves. However, at best, this approach affords only the equivalent of evaluating individual test cases, which cannot guarantee the exploration of important corner cases, nor always eliminate unexpected behavior.

In some sense, a hybrid system can be more challenging to model than a software-only system. A software-only system exists as a logical entity that can be directly translated, without loss of detail, into a representation useful for modeling. A hybrid system description needs to carefully incorporate the continuous behavior of physical components into the system's model.

There have been a series of developments recently, allowing us to apply formal methods to model and rigorously reason about hybrid systems, both in the context of model-checking[1] and deductive verification.[2, 3] In this article, we will describe a hybrid logic called differential-dynamic logic (d$\mathcal{L}$).[4] This logic allows us to rigorously prove, using sound inference rules, properties about the behavior of a hybrid system for all possible inputs.

We will walk through an example of creating a model so that we can apply d$\mathcal{L}$ to certify the safety of a control algorithm for a surgical robot. In this article, our main emphasis is on the process of building a model, which is a central step to proving a hybrid system's safety. We approach building a model with a multistep process that begins with a simplified model, to which we progressively add detail. We express this model as a "hybrid program" (HP), using the programming language available in d$\mathcal{L}$. As we add detail, we will highlight common idioms and strategies for modeling using this approach.

SKULL-BASE SURGERY ROBOT

The target of this analysis is an experimental robot developed by the Johns Hopkins University Center for Integrated Surgical Systems and Technology Group.[5] It was designed to support physicians by helping them move and navigate precisely during surgical procedures on the base of a patient's skull.

The surgical robot was created by integrating three subsystems: a StealthStation navigation system that tracks the position and orientation of sets of optical markers on a rigid body; 3DSlicer, software used for visualizing and analyzing medical image data; and a Neuromate robot, a Food and Drug Administration (FDA)-cleared image-guided 6-degree-of-freedom robotic arm designed for use in neurosurgery.

MODELING A SURGICAL ROBOT CONTROL ALGORITHM

The skull-base surgery (SBS) robot is an example of a cooperatively controlled system. Both the robot and the physician simultaneously hold a surgical tool, and as the surgeon puts force on the tool, the robot allows it to move according to the equation

$$\frac{d\bar{p}}{dt} = \mathrm{G}(\bar{f}), \tag{1}$$

where $\bar{f}$ is the force exerted by the physician, and $\bar{p}$ is the position of the tool in space. The overall equation is a continuous differential equation that represents the effect of a negative feedback control circuit with an admittance control design. This sort of control circuit is designed to convert forces and torques to velocity and is described in Ref. 5. We assume that the admittance control circuit is designed with a sufficiently quick and accurate response so that its imperfections are negligible compared to the response of the system as a whole. The function G is the discrete part of the system; it represents the software that sets the parameters to control the system.

The control algorithm we are evaluating is designed to prevent the surgical tool from crossing a preoperatively defined planar boundary, so as to prevent the tool from moving into an undesirable location. Most of the time, the surgeon can move the tool freely around the surgical site, without interference from the robot. In this region, called the free zone, G is some constant multiple of $\bar{f}$. However, as the tool approaches the boundary, it enters a "slow zone," defined as all points that are within a distance, D, of the boundary, where the component of its velocity toward the boundary is attenuated in proportion to its proximity to the boundary. Eventually, as the tool reaches the boundary, the component normal to the boundary goes to zero, and it should not progress farther in that direction. This is illustrated in Fig. 1.

If the velocity of the tool is $\bar{p}'$, the distance from the boundary is d, and the unit normal to the boundary is $\hat{n}_1$, the overall velocity $\bar{p}'$ is given in Ref. 6 as

$$\bar{p}'_1 = \bar{p}' - \left(1 - \frac{d}{D}\right)(\bar{p}' \cdot \hat{n}_1)\hat{n}_1, \tag{2}$$

where $\cdot$ is the dot product of two vectors, and a prime indicates a derivative with respect to time. This is a subtractive control law, which we will use throughout this

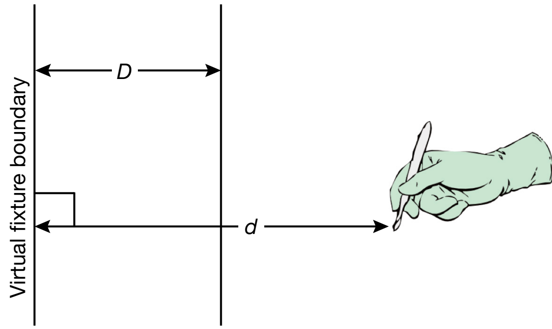

Figure 1. Cooperatively controlled robot enforcing virtual fixtures to restrict a tool-tip to remain on one side of a boundary.

article, even though more advanced controls turn out to be necessary.[7]

To create a model of a cyber-physical system by using d$\mathcal{L}$, we write a HP in d$\mathcal{L}$ modeling language. The language in which we write HPs to model cyber-physical systems is not designed to be executed but to be formally reasoned about; its syntactic constructs are simpler than what we find in executable languages today and are shown in Table 1.

The HP language used in d$\mathcal{L}$ contains arithmetic operators, assignments, and sequential composition of statements and assertions—operations that are found in many imperative languages. State variables in the program can have types that are discrete, finite sets, or R, the reals.

To write a conditional statement in this language, we use an assumption written as a question mark followed by a logical predicate that describes the condition we wish to hold, e.g., $?\chi$. Therefore, we might write $?(a > 5)$; $b := 3$;, which describes a program that can only run when $a > 5$ and then sets b to 3. What happens if $a \leq 5$ in the above program? This program fragment simply cannot execute under these conditions. This is very different from a conventional programming language where the program halts when there is an assertion that is not met. In the case of our HP, an assumption in the middle of the program can noncausally prevent the program from ever executing under those conditions. Because we are not executing the program, but reasoning about it, this does not require unwinding reality; it is simply the equivalent of eliminating certain executions from analysis (e.g., because they cannot be found in the real system).

The HP modeling language used in d$\mathcal{L}$ also has statements that represent the interaction of the program with the physics of a device. To represent these interactions, d$\mathcal{L}$ includes representations of nondeterminism and the ability to describe the evolution of continuous time as a statement in the model. Nondeterminism, for the purposes of d$\mathcal{L}$, is uncertainty that is not described with a probability density function. Conclusions we draw about the system need to be absolute. Thus, theorems we prove must hold for all possible nondeterministic choices, not on the average or a proportion of the time. There are three constructs in the language that can represent nondeterminism: (*i*) assignment, (*ii*) choice, and (*iii*) looping. Assignment of a nondeterministic value to a state variable is written $a := *$. This can assign any value to a. The language also includes a nondeterministic choice of two branches, or alternate behaviors. We write this using the $\cup$ operator. Thus, $(a := 3 \cup b := 6)$ represents a program that can make either of two assignments. Again, the distribution with which it conducts this process is not meaningful because whatever we prove about it must hold for all executions. A conventional "if" statement can be constructed by combining assertions with nondeterministic choice, e.g., $(?(a > 5); b := 3) \cup ?(a \leq 5)$. An indeterminate loop is described with an asterisk appended to the program which is looping. The program $(i := i + 1)*$ can execute the increment instruction some finite number of times before terminating. Again, we can construct conventional control structures such as terminating for and while loops using conditionals and this looping construct, but this indeterminate execution provides us with more expressive flexibility with which to model our programs.

The most interesting statement in a d$\mathcal{L}$ HP is the dynamics statement, which represents the continuous physics of an analog device. Such a statement contains a system of differential equations, and its presence indicates continuous evolution of state variables in a manner that provides a solution to these equations, as long as the state satisfies the logical constraint, χ:

$$x'_1 = \theta_1, \ldots, x'_n = \theta_n \& \chi. \quad (3)$$

We can think of dynamics statements as allowing time in the real world to progress and whatever physics governs our system to operate. The rest of

Table 1. d$\mathcal{L}$ modeling language

Statement	Description
$x_i := \theta_i$	assignment
$x_i := *$	nondeterministic assignment assigns any value
$\alpha; \beta$	sequence first runs α, then β
$?\chi$	assumption
$\alpha \cup \beta$	nondeterministic choice
$\alpha *$	nondeterministic loop
$x'_1 = \theta_1, x'_2 = \theta_2, \ldots, x_i := \theta_i \& \chi$	dynamics statement describing continous evolution

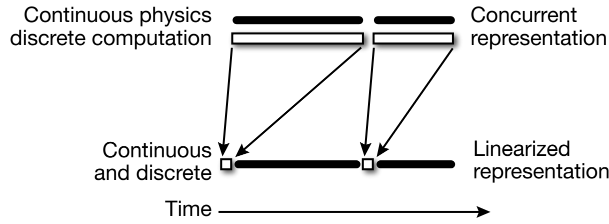

Figure 2. Illustration of the HP strategy for representing the concurrent interaction between the evolution of continuous physics and discrete computation.

the statements describe discrete computations and state changes inside a computer and have no real time associated with them. It is as if when discrete statements are encountered in sequence, the continuous dynamics of the system are "frozen," and when the system encounters a dynamics statement, discrete computation ceases and the continuous dynamics evolve.

Despite the apparent stop-and-go nature of the dynamics in this model, it is effective at modeling a system that is doing discrete computations as the continuous dynamics evolve. In the real world, discrete computations and continuous dynamics operate concurrently, but the HP forces us to linearize them, putting them into a sequential order. We conceptually associate the period of time in which the physics evolved to be concurrent with the discrete computation that preceded it in the program. Within the HP, the discrete program and the continuous physics can share state at the edges of these linearized transitions, as shown in Fig. 2.

In the rest of this article, we will develop an algorithm that enforces a virtual boundary that is arbitrarily oriented and, when encountered by the user, qualitatively feels hard. In other words, an encounter between the tool and the boundary will abruptly stop the tool, preventing it from crossing but allowing it to slide along the boundary, giving it a slippery feel. This control algorithm differs qualitatively from the one described in Ref. 7, which provides force feedback, making the walls feel like they are soft, leading to a gradual stop, and where contact with the boundary feels sticky. Our first model will be a simplified version, to which we will add detail.

Simple Model

Consider the control algorithm implemented to constrain the movement of a tool, whose dynamics are

$$\frac{dp}{dt} = \mathrm{G}(f), \tag{4}$$

where G represents the effect of the software that we write for the system, f is the force in the direction of the axis, and p is the position of the tool.

For the initial model, we will constrain the tool in one direction, the x axis, so it restricts movement by providing a hard stop on only one side. In this simplified scenario, p is a positive value representing the tool's position on the x axis. Assume the stop is set to position $p = 0$, and we wish to design an algorithm that ensures that the sled comes to an abrupt halt, bouncing off of a virtual boundary that feels rigid.

If we are sufficiently far from the track's end at $p = 0$, we would like to allow the system to move freely, translating force into velocity via a constant coefficient, g. When we are close to the track's end, we wish to create a buffer that attenuates its motion in proportion to how close it comes to the end. For simplicity, we can assume that the force, f, the surgeon exerts on the tool is constant over a short time period. We can write a HP that describes the different states of the system by describing each state with a dynamics statement:

$$\begin{aligned}
&\mathrm{ctrl} = (f := *;\\
&(p' = gf \,\&\, f >= 0) \cup\\
&(p' = gf \,\&\, (f <= 0) \wedge (p >= D)) \cup\\
&(p' = g(p\,/\,D)f \,\&\, (f <= 0) \wedge (p <= D)))*
\end{aligned} \quad . \tag{5}$$

The first line of this HP is a nondeterministic assignment of some value to the state variable f. It represents the surgeon's arbitrary input. The HP is a loop, whose body comprises a nondeterministic assignment followed by a nondeterministic choice between three different dynamics statements, each of which is given on its own line. These dynamics statements describe the evolution of our system according to the differential equations given in each case, which can only occur when the specified constraints $f >= 0$, $(f \le 0) \wedge (p \ge D)$, or $(f \le 0) \wedge (p \le D)$ are satisfied.

The outer loop (i.e., $*$) represents the permanent operation of the system. For each time through the loop, there is some constant force input represented with the nondeterministic assignment, and this is followed by a nondeterministic choice. Because the HP in Eq. 5 must choose one of these, it chooses one where the constraints on the right side of the dynamics statement are satisfied and solves the differential equation given. The tool moves along at a speed proportional to how hard it is pushed, with the constant of proportionality either given by g or by $p\,/\,D$. The system can switch state at any time, if the constraints allow it.

Proving Safety of the Simplified Model of a Control Algorithm

To describe the desired behavior of our model, we construct logical predicates describing the relationships between different state variables, using the operators given in Table 2. These predicates can contain operators from first-order logic (e.g., $\phi \wedge \psi$ represents logical "and") and quantifiers (e.g., $\forall\ x\ \phi$ represents the state-

Table 2. Different operators available in d$\mathcal{L}$ to express behavior

$a \wedge b$	logical and
$a \vee b$	logical or
$\neg a$	negation
$\exists x P(x)$	existential quantification over the reals
$\forall x P(y)$	universal quantification over the reals
$[\alpha]\chi$	box modality: all runs of α satisify the postcondition χ
$\langle\alpha\rangle\chi$	angle modality: there exists a run of α that satisifies the postcondition χ

ment that ϕ holds for all values of x). We also can write modal operators (e.g., $[\alpha]\phi$ means that after running the HP α, the predicate ϕ always holds). The modal operators are very powerful because they allow a sort of quantification over different executions of a given program, and they are at the core of how we describe properties of hybrid models of cyber-physical systems.

We write the safety property we desire to prove for our algorithm as

$$(p >= 0) \rightarrow [\text{ctrl}](p >= 0). \quad (6)$$

This predicate is called our goal, or the theorem that we wish to prove. It says: if the tool starts in a safe place (i.e., $p >= 0$) and you run the control program, then the tool will remain in a safe place at every instant of time, no matter what input you provide for it.

To do the proof, we apply sound inference rules to our goal, decomposing it into simpler goals and eventually statements in real arithmetic. This real arithmetic can be solved by a computer program called a decision procedure. This proving process is the heart of d$\mathcal{L}$; the logic is constructed so that completing such proofs translates into a guarantee about the system's behavior, for all possible system inputs.

The proof will be structured the same way as the model, first decomposing the outer loop into three different subgoals: the base case, the inductive step, and the postcondition. Within the proof of the inductive step, there are three cases, one for each possible nondeterministic choice represented by the dynamics statements. The differential equation must be solved for each one, and the proof about the inductive step can be completed.

KeYmaera[8] is a tool that takes a HP and allows an engineer to create a machine-checked proof by using d$\mathcal{L}$. This is called a mechanization of the logic. This system is simple enough that once the HP model and safety property are entered, a loop invariant can be provided, and safety for this simplified 1-D system can be proved automatically at the press of a button. A loop invariant is a logical statement that describes an important, unchanging attribute that will hold at the beginning and end of a loop; it is necessary to resolve the behavior of complicated loops automatically.

A Generalized Model of a Single Virtual Fixture Boundary

The simplified model represents input as a constant during a time step and does not accurately represent a lag in the system. It assumes that the program that implements the controller runs all the time. It also fixes the position of the virtual boundary and provides control in only one dimension. Although these simplifying assumptions were a useful starting point, we need to replace them with more realistic assumptions to ensure that our conclusions are true. This section refines the previous model to relax these assumptions and create a more realistic controller in two dimensions.

The first enhancement to the model will be a more accurate representation of user input. The previous model represented user input for each "step" in the system as a constant, by setting the force components to a nondeterministic value by writing $f_x := *; f_y := *$. A more accurate model of the force input to the system would be to create a piecewise linear representation of the force. To do this, we assign nondeterministic values to some state variables, $f_{xp} := *; f_{yp} := *$, and then logically associate these with the derivative of the force by requiring $f_x' = f_{xp}$ and $f_y' = f_{yp}$ in the set of differential equations that we specify during continuous evolution. When we make this change, it ensures that at each step, we have a constant acceleration. This makes our velocity piecewise linear and our path quadratic, given the simple relationship between these state variables. Consequently, there are many additional possible types of force curves that can be exerted on the system during a time step, depending on the acceleration and the initial direction of the force during the step. We can distinguish between these different types of curves and design our controller to recognize and behave differently in each situation. The different movement scenarios are shown in Fig. 3. Each subfigure represents a movement scenario in which the SBS robot must enforce safety. Each case must start above the x axis because the system starts in a safe location. The different cases represent different accelerations, velocities, and possibilities for intersecting or not intersecting the boundary. If they do intersect, the controller needs to take action to ensure safety. The final, safe version of the realistic controller contains each of these scenarios as an explicitly controlled case.

The second enhancement is to add a clock to our system to track the passage of time and use it to represent the fundamental loop used during the control algo-

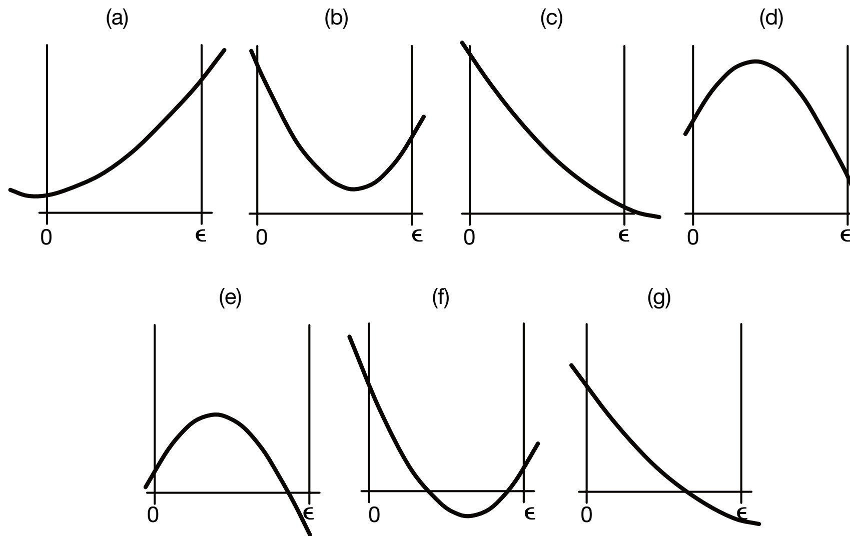

Figure 3. Different movement scenarios in which the SBS robot must enforce safety. The y axis of each subfigure is the perpendicular distance from the tool to the boundary [i.e., $d(t)$], while the x axis represents the progression of time.

rithm. In d$\mathcal{L}$, this is a relatively simple matter: we add a state variable, t, to the program to hold the value of the clock at any instant of time; before each dynamics statement, we reset our time counter $t := 0$; within each dynamics statement, we require that $t' = 1$ (i.e., time progresses continuously at a rate of 1); and we put a constraint on the evolution in each dynamics statement to ensure that a time step remains below a certain threshold, $t \leq \epsilon$, to represent a step in the control algorithm. In this case, ϵ represents the largest time stop the controller can take before responding to its environment.

Original HP	**HP with clock**
(discrete;	(discrete;
dyn)*	$t := 0$; (dyn, $t' = 1 \& t \leq \epsilon$))*

This is a common idiom for representing time in a d$\mathcal{L}$ HP. The left program is the original, containing discrete computation and a dynamics statement to allow the evolution of continuous physics. The program on the right is a transformation of the program on the left, showing how a time reference can be added to it. The time reference can be used to restrict the length of the dynamics statements, as well as used in the discrete program's computations.

The third enhancement is to more accurately represent time lag associated with the discrete program's execution on the basis of the time it takes to execute the computations necessary to make a control decision. In its simplified form, our model in Eq. 5 has a different dynamic statement for each possible mode of operation, whose functional form is tailored to that state. While this is a reasonable starting point for our model, it is not very realistic; we are acting as if we have a discrete program that can instantaneously run and modify the parameters of the system at every moment in time.

This problem would quickly become evident if we tried to represent a more complex controller by using this approach. We would encounter difficulties even creating such a representation; for example, if we needed a looping construct to represent our control logic, we would need to put it in our dynamics statement, mixing it with the continuous differential equation there. This is not allowed by d$\mathcal{L}$ because it is not realizable.

To represent a more realistic program, we will factor the discrete computation for each mode out of the dynamics statement and create a single dynamics statement that represents physics and the evolution of time. The collection of discrete computations and constraints that we factor out of the dynamics statements represents the control algorithm setting parameters and exerting control over the continuous dynamics once at every step in the system. Thus, we can accurately represent time lag associated with a program and its control loop:

Instantaneous	**Noninstantaneous**
ctrl ≡	ctrl ≡
(discrete;	(discrete;
(mode1 dyn) ∪	(mode1discrete ∪
(mode2 dyn) ∪	mode2discrete ∪
(mode3 dyn))*	mode3discrete);
	dyn)*

This is a common idiom for representing realistic, noninstantaneous execution of discrete computation. The program on the left is an idealized version of the algorithm that responds instantaneously. The program on the right is a transformation of the program on the left, showing how discrete computation may be factored out of the continuous physics and consolidated to add more realism to the model.

Once this is done, there will be only one set of differential equations that describes the continuous evolution of our system, and it will simply represent the physics of the system and the behavior of our lower-level admit-

Table 3. A complete time-triggered model of a redesigned control algorithm that enforces the safety of an arbitrary number of arbitrarily oriented and positioned virtual fixture boundaries, in three dimensions (this model is realistic, provides directional force feedback, and is proven to be safe, via formal, mechanized proof)

$$
\begin{aligned}
&\mathrm{ctrl}_2 \equiv \Big(\\
&\quad f_{xp} := *; f_{yp} := *; \\
&\quad f_{np} := (f_{xp} n_x + f_{yp} n_y); \\
&\quad f_n := (f_x n_x + f_y n_y); \\
&\quad d_0 := ((q_x - px) n_x + (q_y - py) n_y); \\
&\quad dist := (d_0 + K(f_n\, e + (f_{np} e^2)/2)); \\
&\quad disc := ((K f_n)^2 - 2K f_{np} d_0); \\
&\quad (?f_{np} \le 0 \wedge dist \ge 0; \\
&\quad\quad g := 0) \\
&\quad \cup(?f_{np} \le 0 \wedge dist \le 0; \\
&\quad\quad g := (f_n + (d_0 + (K f_{np} e^2)/2)/(Ke))) \\
&\quad \cup(?f_{np} \ge 0 \wedge f_n \le 0 \wedge disc \le 0; \\
&\quad\quad g := 0) \\
&\quad \cup(?f_{np} \ge 0 \wedge f_n \le 0 \wedge disc \ge 0 \wedge f_n + f_{np} e \ge 0; \\
&\quad\quad g := (f_n - Sqrt((2 d_0 f_{np})/K))) \\
&\quad \cup(?f_{np} \ge 0 \wedge f_n \le 0 \wedge disc \ge 0 \wedge f_n + f_{np} e \le 0 \wedge dist \le 0; \\
&\quad\quad g := (f_n - Sqrt((2 d_0 f_{np})/K))) \\
&\quad \cup(?f_{np} \ge 0 \wedge f_n \le 0 \wedge disc \ge 0 \wedge f_n + f_{np} e \le 0 \wedge dist \ge 0; \\
&\quad\quad g := 0) \\
&\quad \cup(?f_{np} \ge 0 \wedge f_n \ge 0; \\
&\quad\quad g := 0); \\
&\quad t := 0; \\
&\quad (q_x' = K(f_x - g n_x), q_y' = K(f_y - g n_y), f_x' = f_{xp}, f_y' = f_{yp}, t' = 1 \,\&\, t \le e) \\
&\Big)^*
\end{aligned}
$$

tance controller, which should not change regardless of the mode the system is in or the damping decision made by the control algorithm.

A more realistic model of our control algorithm that includes these enhancements is shown in Table 3. This algorithm enforces a single virtual boundary based on the subtractive control law given in Eq. 2. The subtractive control law works by modifying the overall velocity by subtracting the part of the vector out of the movement that is normal to the virtual boundary. This boundary should feel abrupt; it is encountered with no warning, so the surgeon needs another way of visualizing how close to it he or she is. When pressed against it, the tool will tend to bounce back slightly, giving the boundary a bouncy feel. This comes from a combination of the subtractive control law and the system's delay.

The user should not experience any of the stickiness we expect from the control algorithms produced by multiplicative damping strategies (i.e., those strategies that modify the overall velocity by applying a multiplicative factor less than 1), but instead the virtual boundary will feel somewhat slippery because of the selective removal of the movement component in the direction of the boundary.

Proving Safety of an Enhanced Model of a Control Algorithm

We implemented the realistic model of our controller and proved its safety using KeYmaera. We can trust KeYmaera because it faithfully implements the inference rules in d$\mathcal{L}$, which are guaranteed to yield correct conclusions because they themselves have been proven to be sound.[4] The proof of safety for our controller was created by a mixture of automated steps that decomposed the HP according to its structure, manual steps used to guide the proof for each of the different branches, and quantifier elimination to discharge real arithmetic at the leaves of the proof tree.

The proof is structured like the program with an initial set of three branches relating to the loop and its invariant, namely proving a necessary precondition, a desired postcondition, and the inductive step. In the inductive step, there are seven major branches, one for each of the different input cases we encounter. Figure 4 shows the major branches of this proof, in the leftmost pane, as a hierarchical tree. The green folders marked "Case 1" are all major branches corresponding to a movement case, and the last "Case 2" branch is the last movement case. KeYmaera highlights completed proof branches with green to indicate that their proof is completed and mechanically checked. The right pane of the proof window shows the state of the proof, with a sequent that contains assumptions and the goal to be proved at the intermediate state highlighted in blue on the left.

Figure 5 shows a completed, machine-checked proof that guarantees that our algorithm safely enforces a virtual planar boundary under all possible input conditions. The proof has a total of 1582 nodes and 163 branches. The various branches are generated from the different circumstances in which the system can find itself, and the different nodes are created each time an inference rule is applied to transform our proof state. It takes 19 minutes for a standard laptop to machine-check the completed proof.

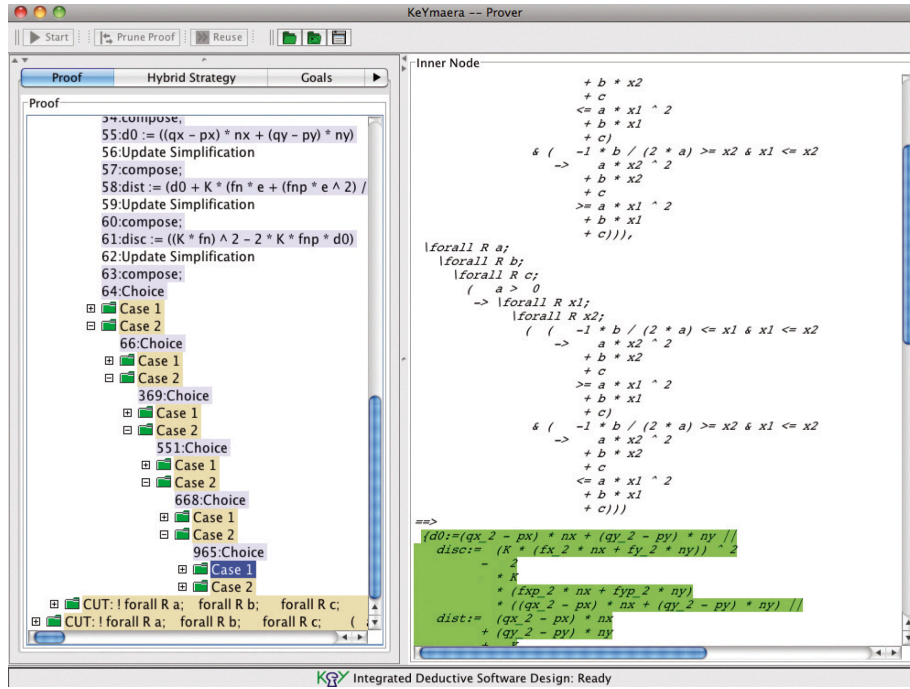

Figure 4. KeYmaera tool displaying intermediate proof state and major branches for surgical robot control algorithm safety proof.

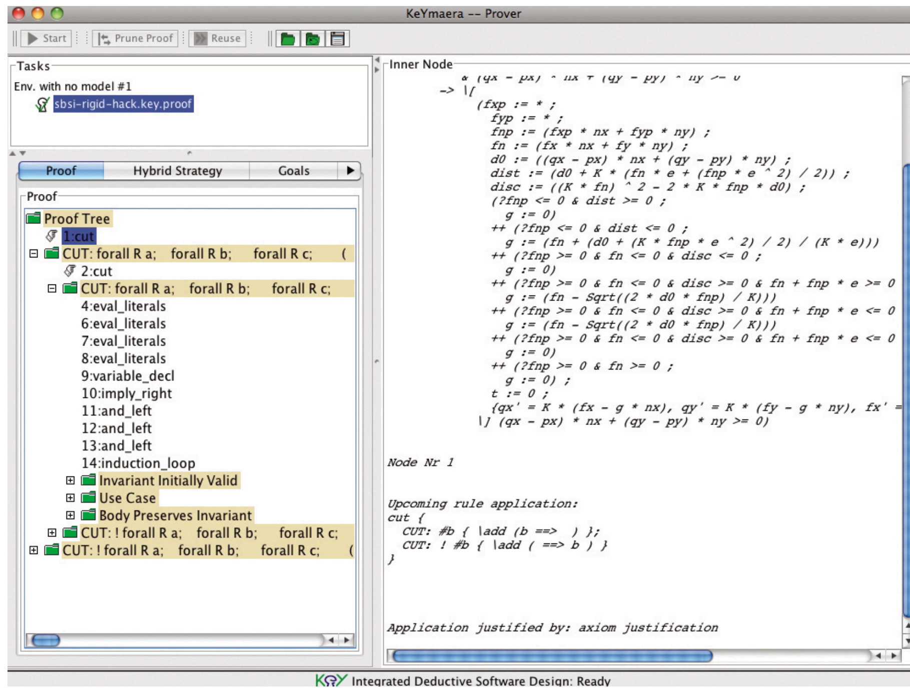

Figure 5. KeYmaera tool displaying completed proof of safety for surgical robot control algorithm.

FURTHER WORK

As part of this robotic control formal verification, we further developed the 2-D model described in the preceding section to increase its fidelity. We created a 3-D model of a virtual fixture boundary defined by a set of arbitrarily oriented planes to constrain the tool, as shown in Fig. 1. Additionally, instead of implementing a hard stop, we developed an algorithm that created a soft stop where the force required to move the blade increased in proportion to the nearness to the boundary so that the tool came to a stop and could not slide across

the boundary. This work is detailed in Ref. 7 and was proved for arbitrarily many boundaries.

CONCLUSION

We have powerful tools available today to reason about cyber-physical systems. With them, we can create an accurate model of a cyber-physical system component and make guarantees about the system's behavior under all possible input conditions.

These tools are new, and it is sometimes difficult to apply them to larger, more complicated system components because the proof becomes commensurately more complex.

Along with investigating many other advances in proof automation in KeYmaera, we are in the process of exploring how to effectively scale these capabilities to larger systems and how to compose proofs about small system components together to make guarantees about larger subsets of the system.

REFERENCES

[1]Fränzle, M., and Herde, C., "HySAT: An Efficient Proof Engine for Bounded Model Checking of Hybrid Systems," *Formal Methods in System Design* **30**(2), 179–198 (2007).

[2]Platzer, A., "Differential-Algebraic Dynamic Logic for Differential-Algebraic Programs," *J. Log. Comput.* **20**(1), 309–352 (2010).

[3]Platzer, A., "A Complete Axiomatization of Quantified Differential Dynamic Logic for Distributed Hybrid Systems," *Logical Methods in Computer Science* **8**(4), 1–44 (2012).

[4]Platzer, A., "Differential Dynamic Logic for Hybrid Systems," *J. Autom. Reasoning* **41**(2), 143–189 (2008).

[5]Xia, T., Baird, C., Jallo, G., Hayes, K., Nakajima, N., et al. "An Integrated System for Planning, Navigation and Robotic Assistance for Skull Base Surgery," *Int. J. of Med. Robot.* **4**(4), 321–330 (2008).

[6]Kazanzides, P., "Virtual Fixture Computation. Note on Combining the Effects of Multiple Virtual Fixtures" (2011). [Please contact author to obtain a copy of document.]

[7]Kouskoulas, Y., Renshaw, D., Platzer, A., and Kazanzides, P., "Certifying the Safe Design of a Virtual Fixture Control Algorithm for a Surgical Robot," in *Proc. 16th International Conf. on Hybrid Systems: Computation and Control*, pp. 263–272 (2013).

[8]Platzer, A., and Quesel, J.-D., "KeYmaera: A Hybrid Theorem Prover for Hybrid Systems," in *Automated Reasoning: 4th Int. Joint Conf.*, Vol. 5195 of LNCS, A. Armando, P. Baumgartner, and G. Dowek (eds.), SpringerVerlag, Berlin Heidelberg, pp. 171–178 (2008).

The Authors

Yanni Kouskoulas is a group chief scientist at APL. His research interests are focused on formal methods and theorem proving and their application to help develop zero-defect software for a variety of practical cyber-physical systems. **André Platzer** is an assistant professor of computer science at Carnegie Mellon University. His interests include logic in computer science, cyber-physical systems, programming languages, formal methods, and theorem proving. He received an Association for Computing Machinery Doctoral Dissertation honorable mention and an National Science Foundation CAREER Award for his work in this area. He developed differential-dynamic logic and the KeYmaera tool, which were used for modeling and theorem proving during this work. **Peter Kazanzides** is an associate research professor of computer science at The Johns Hopkins University. His research interests include computer-integrated surgery and systems engineering to enable deployment in the real world. While working at Integrated Surgical Systems, he was responsible for the design and implementation of the ROBODOC system, which has been used for more than 20,000 hip and knee replacement surgeries. He helped develop the surgical robot control algorithm that was the subject of analysis during this work. For further information on the work reported here, contact Yanni Kouskoulas. His e-mail address is yanni.kouskoulas@jhuapl.edu.

The *Johns Hopkins APL Technical Digest* can be accessed electronically at **www.jhuapl.edu/techdigest**.