Towards Verification of Hybrid Systems in a Foundational Proof Assistant

Daniel Ricketts Gregory Malecha Mario M. Alvarez Vignesh Gowda Sorin Lerner
University of California, San Diego
La Jolla, California 92037
Email: {dricketts,gmalecha,mmalvare,vngowda,lerner}@cs.ucsd.edu

***Abstract*—Unsafe behavior of hybrid systems can have disastrous consequences, motivating the need for formal verification of the software running on these systems. Foundational verification in a proof assistant such as Coq is a promising technique that can provide extremely strong, foundational, guarantees about software systems. In this paper, we show how to apply this technique to hybrid systems. We define a TLA-inspired formalism in Coq for reasoning about hybrid systems and use it to verify two quadcopter modules: the first limits the quadcopter's velocity and the second limits its altitude. We ran both of these modules on an actual quadcopter, and they worked as intended. We also discuss lessons learned from our experience foundationally verifying hybrid systems.**

I. Introduction

From 400-seat commercial airplanes to miniature hobbyist quadcopters, hybrid systems surround us. While this opens up exciting possibilities, it also presents enormous risks. Unsafe behavior of a hybrid system can have consequences ranging from the loss of costly equipment to the loss of human life. The software running on hybrid systems offers the potential to prevent dangerous behaviors; for example, an aircraft can have a software module that intervenes if the pilot attempts to perform an unsafe maneuver. However, since it performs such a critical function, it is important to guarantee that this software is correct. Formal verification can provide such a guarantee.

Foundational verification is a particular kind of verification technique that has shown increasing promise in the past decade [1]. In this technique, the programmer writes programs in a proof assistant such as Coq [2], and then interactively proves these programs correct. The main benefit of foundational verification is that it provides a highly-detailed correctness proof that can be checked using a very small, trusted, proof checker. Experimental studies have shown that foundationally verified software is far more reliable than software written in a traditional way [3]. While there has been a tremendous amount of work on formal verification of hybrid systems [4]–[6], there has been less work on foundational verification of hybrid systems [7]–[9].

In this paper, we show how to apply deductive, foundational verification to hybrid systems. Broadly speaking, foundational verification is challenging because it requires all details to be worked out in full detail. Often, formalizing systems at this level of detail uncovers subtleties and mistakes that are difficult to see otherwise. Throughout the paper we will present examples of this kind of subtlety that we uncovered in developing our work.

In our work we define a TLA[1]-inspired formalism for reasoning about hybrid systems in Coq [2]. Our formalism allows us to express both hybrid systems and properties of these systems in a uniform way, as well as to use logical rules to prove the stated properties. We use our formalism to implement and verify two quadcoptor modules: the first limits the velocity of the quadcopter, and the second limits its height. These modules can be added to any controller as a small runtime monitor, or *shim*, between the controller and the motors. These shims adjust the values sent to the motors to make sure that certain safety properties are preserved. In addition to verifying our shims, we also ran them on a flying Iris+ quadcopter, and both shims worked as intended.

In summary, our contributions are the following:

- We demonstrate techniques for performing foundational verification of hybrid systems in Coq. We provide an overview of our approach (Section II), followed by a more formal description of the logic (Section III).
- We use our approach to implement and verify two shims (Section II), which we installed and flew on a real quadcopter (Section IV). The full development is part of the VeriDrone project, available at http://goto.ucsd.edu/veridrone/.
- We discuss lessons learned from our development (Section V), which we hope will help and guide future research in this direction.

II. Verifying Shims

We start by giving an overview of how to apply our techniques to verify shims from the domain of Unmanned Aerial Vehicles (UAVs). For instance, we might want a guarantee that the software running on a UAV prevents it from climbing more than 400 feet above the ground (and thus violating FAA regulations). However, we would like to accomplish this while allowing the UAV to run complex controllers that may not guarantee these properties.

We accomplish this by implementing and verifying a simple safety *shim*, which runs after all of the (potentially complex) controllers have run, but before any signals are sent to the motors. This shim performs a safety check on the outputs from the higher level controllers. In our height shim example, the shim checks whether the output of the higher

[1]Lamport's Temporal Logic of Actions [10]

978-1-5090-0237-5/15/$31.00 ©2015 IEEE

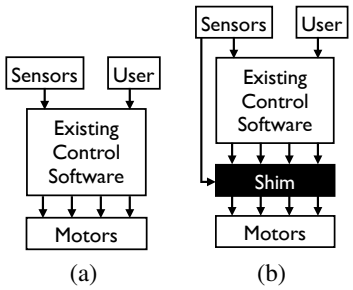

Fig. 1. A simplified depiction of UAV architecture (a) without and (b) with our shim.

level controllers would put the UAV in a state in which it could not stop before exceeding the upper bound. If the check passes then the shim issues exactly the same outputs as the higher level controllers. Otherwise, the shim issues a conservative action to ensure the desired safety property.

Our architecture, inspired by the simplex architecture [11], is depicted in Figure 1. This architecture allows us to focus our verification effort on the safety critical shim, without needing to reason about complex existing controllers. This design makes the verification tractable, while retaining the benefits of state-of-the art controllers.

Our verification of hybrid systems in Coq is built on top of a discrete-time linear temporal logic (called RTLA) inspired by Lamport's Temporal Logic of Actions (TLA) [10], a formalism of temporal logic designed with refinement in mind. Section II-A contains a brief overview of RTLA. Next we describe our Sys abstraction for modeling hybrid systems in RTLA (Section II-B). We then demonstrate how we specify, verify, and compose several hybrid systems using our abstraction.

A. Background: TLA

RTLA formulas specify traces (discrete sequences of states) allowed by a system. We briefly illustrate RTLA through the following example which specifies a system that repeatedly increments the variable x by 1:

$$x = 0 \wedge \Box(x' = x + 1)$$

The initial condition, $x = 0$, is called a state formula and characterizes all possible initial states of the system, stating that x must be 0 in those states. This initial condition corresponds to an infinite number of states since all other variables, e.g. y and z, are unconstrained.

The second part of the above RTLA formula describes how the system evolves. The expression between the parentheses is an *action formula* specifying the relationship between two temporally adjacent states. In RTLA, x' denotes the value of x in the next state. Thus, the formula states that the value of x in the next state is 1 greater than in the previous state. This relation about two adjacent states is lifted to a statement about traces (sequences of states) by the "always" operator ($\Box$), which states that all future temporally adjacent pairs of states are related by the given action formula.

Note that RTLA is a discrete-time temporal logic; there are no continuous evolutions in RTLA traces. When specifying hybrid systems, which contain continuous evolutions, an RTLA formula characterizes all possible sequences of samplings of the system state. We further discuss and justify this specification in Section III-B.

B. Our Hybrid System Model

We now turn to our Sys abstraction for describing hybrid systems. Sys takes 4 parameters:

$$\mathsf{Sys}\ I\ (\mathcal{W}, C)\ D\ \Delta$$

where the parameters are as follows:

- I — an RTLA state formula describing the initial condition
- $(\mathcal{W}, C)$ — a pair describing the continuous dynamics of the world. $\mathcal{W}$ is a set of differential (in)equations, and C is an RTLA action formula stating additional constraints about how the world evolves.
- D — an action formula specifying the discrete controller
- Δ — a constant giving the maximum time between two executions of the discrete controller

$\mathsf{Sys}\ I\ (\mathcal{W}, C)\ D\ \Delta$ unfolds to an RTLA formula, but we leave the full details of this formula to Section III-B. Informally, for now, Sys is defined in such a way that the system starts in a state satisfying I and can repeatedly take either of two kinds of transitions: an instantaneous transition of the discrete program (D); or a continuous evolution of the physical world described by $\mathcal{W}$ under the constraints in C.

We now show how Sys can express a shim that ensures the velocity of a simple one-dimensional model of a quadcopter never exceeds a constant upper bound $\mathsf{v_{ub}}$. Our shim makes use of the following variables (by convention, lower case variables are continuous and upper case variables are discrete): v is the actual velocity of the system, whose behavior will be specified by differential equations encoding the physics of the real world; v_{max} is an upper bound on v (e.g. produced by a sensor) and is an input to our shim; $T_?$ is the thrust requested by the higher level controller, which is also an input to our shim; T is the thrust produced by our shim, which gets sent to the motors. Our shim is defined as follows:

$$\mathsf{VelShim} \equiv \mathsf{Sys}\ I\ (\mathcal{W}, C)\ D\ \Delta$$

where

$$\begin{array}{rcl} I & \equiv & \max(0, T - g) \cdot \Delta + v \leq \mathsf{v_{ub}} \\ \mathcal{W} & \equiv & \dot{v} \leq T - g \\ C & \equiv & \mathsf{True} \\ D & \equiv & ((T_? - g) \cdot \Delta + v_{max} \leq \mathsf{v_{ub}} \wedge T'{=}T_?) \vee T'{=}g \end{array}$$

I states that the velocity is at most $\mathsf{v_{ub}}$ and furthermore is small enough that it will still be at most $\mathsf{v_{ub}}$ when the shim first runs (which will be at most Δ time units away). $\mathcal{W}$ states the differential inequality capturing how velocity is related to thrust (T) and gravity (g). This relationship is an inequality rather than an equality because we are modeling a quadcopter, whose vertical thrust is only upper-bounded by the thrust produced by the motors. We discuss this further in Section IV. The RTLA action formula D captures the control logic of our shim. The disjunction encodes, essentially, the following conditional statement: $\mathtt{if}\ T_? \cdot \Delta + v_{max} \leq \mathsf{v_{ub}}\ \mathtt{then}\ T' = T_?\ \mathtt{else}\ T' = g$;

though it also allows executing the safe action ($T' = g$) when the check succeeds.

Since Sys unfolds directly to an RTLA formula, we can express the correctness of our shim directly in RTLA as follows (where $\vdash$ represents entailment in RTLA, expressing that the formula to the right of the $\vdash$ holds on all traces satisfying the formula to the left of the $\vdash$):

$$\Box v \leq v_{max} \vdash \mathsf{VelShim} \rightarrow \Box v \leq \mathsf{v_{ub}} \tag{1}$$

This formula states that if v_{max} is always an upper bound on the actual velocity ($\Box v \leq v_{max}$), then the velocity shim ensures that the velocity of the system is always less than or equal to $\mathsf{v_{ub}}$. For now, we do not specify how the value of v_{max} is produced; that is, v_{max} does not appear in any action formulas (transitions). Instead, we simply assume that such a value is provided to the shim. In Section II-D, we will show how to specify systems that produce a v_{max} satisfying this assumption and we will show how to compose them with the velocity shim.

C. Verifying the Velocity Shim

We now show how we prove the above RTLA formula in Coq. To do so, we need an inductive invariant that is preserved both by the continuous transitions (those of the world) and the discrete program. Although we have implemented several mechanisms for simplifying reasoning about Sys, we currently do not infer invariants automatically. To express our invariant, we need a way to track the maximum amount of time that can elapse before the *next* execution of the discrete program. This value is tracked by the τ variable. Using τ, we can express the inductive invariant as follows:

$$\max(0, T - g) * \tau + v \leq \mathsf{v_{ub}}$$

In order to prove that this formula is an inductive invariant, we use the the SysInd proof rule (in Figure 2), which is a special case of discrete induction tailored to systems that are described by our Sys construct. Proof rules such as this one allow us to abstract the implementation of Sys and make for overall cleaner proofs. Informally, SysInd states that a formula P is an (inductive) invariant of a system if P holds on all possible initial states of the system, and P is preserved by the two possible transitions that the system can make. We use P' to denote the formula P with all unprimed variables x replaced with their primed counterpart x'. We use *free* to informally denote a function that takes an RTLA formula and returns the free variables of the formula, and we use *nonzero* to denote a function that takes a list of differential equations and returns a list of variables with non-zero derivatives (i.e. the continuous variables). In the actual formalism, the variables are specified manually. We also use $\mathsf{Unchanged}(X)$, where X is a set of variables, to represent the RTLA formula stating that each variable in X is equal to its primed counterpart.

Aside It is important to realize that all the rules we use (for example those in Figure 2) are foundationally verified with respect to the semantics of RTLA. As we will discuss more in Section V, this level of foundational proof has been useful in uncovering bugs both in our shims and in our formalism.

To illustrate how the proof works, we walk through the proof obligations obtained by applying SysInd to our velocity

$$\frac{\begin{array}{c} c = \mathit{nonzero}(\mathcal{W}) \cup \mathit{free}(C) \quad d = \mathit{free}(D) \\ Q \vdash 0 \leq \tau \leq \Delta \wedge I \rightarrow P \\ Q \vdash 0 \leq \tau \leq \Delta \wedge P \wedge D \wedge \mathsf{Unchanged}(c \cup \{\tau\}) \rightarrow P' \\ Q \vdash \tau \leq \Delta \wedge 0 \leq \tau' \wedge P \wedge \mathsf{World}_d(\mathcal{W}) \wedge C \rightarrow P' \end{array}}{\Box Q \vdash \mathsf{Sys}\ I\ (\mathcal{W}, C)\ D\ \Delta \rightarrow \Box P}\ \textsc{SysInd}$$

$$\frac{\begin{array}{c} \vdash \mathsf{Sys}\ I_a\ (\mathcal{W}, C_a)\ D_a\ \Delta \rightarrow \Box P \\ \Box P \vdash \mathsf{Sys}\ I_b\ (\mathcal{W}, C_b)\ D_b\ \Delta \rightarrow \Box Q \end{array}}{\vdash \mathsf{Sys}\ I_a\ (\mathcal{W}, C_a)\ D_a\ \Delta \circ \mathsf{Sys}\ I_b\ (\mathcal{W}, C_b)\ D_b\ \Delta \rightarrow \Box(P \wedge Q)}\ \textsc{SysCompose}$$

Fig. 2. Rules for reasoning about systems.

shim. First, we must prove that P holds on all initial states of the system:

$$v \leq v_{max} \quad \vdash \begin{bmatrix} & 0 \leq \tau \leq \Delta \\ \wedge & \max(0, T - g) \cdot \Delta + v \leq \mathsf{v_{ub}} \\ \rightarrow & \max(0, T - g) \cdot \tau + v \leq \mathsf{v_{ub}} \end{bmatrix}$$

Proving this requires first order reasoning over real arithmetic in Coq. We can solve simple obligations such as this one, using existing Coq real arithmetic decision procedures [12] that produce foundational Coq proofs completely automatically. While these procedures are not complete, they are still able to discharge many obligations that arise in practice. When they are unable to completely prove a goal, we are forced to manually construct a machine-checked proof of the remaining obligations. We discuss this process in more detail in Section II-E.

Next, we prove that the inductive invariant is preserved by discrete steps of the system. There are actually two cases to prove: when the proposed thrust passes the shim's safety check and when the shim issues a thrust equal to gravity. In the first case, we are left to prove the following proof obligation (the reasoning in the second case is simpler):

$$v \leq v_{max} \quad \vdash \begin{bmatrix} & 0 \leq \tau \leq \Delta \\ \wedge & \max(0, T - g) * \tau + v \leq \mathsf{v_{ub}} \\ \wedge & (T_? - g) * \Delta + v_{max} \leq \mathsf{v_{ub}} \\ \wedge & T' = T_? \\ \wedge & v' = v \\ \wedge & \tau' = \tau \\ \rightarrow & \max(0, T' - g) * \tau' + v' \leq \mathsf{v_{ub}} \end{bmatrix}$$

Note that the velocity is unchanged ($v' = v$) because Sys ensures that continuous variables are unchanged during discrete transitions. Proving this obligation requires first order reasoning over real arithmetic, but fits into the automation described above.

Finally, we prove that the inductive invariant is preserved by continuous transitions. This proof obligation is slightly more difficult:

$$v \leq v_{max} \quad \vdash \begin{bmatrix} & \tau \leq \Delta \wedge 0 \leq \tau' \\ \wedge & \max(0, T - g) \cdot \tau + v \leq \mathsf{v_{ub}} \\ \wedge & \mathsf{World}_d(\mathcal{W}) \\ \rightarrow & \max(0, T' - g) \cdot \tau' + v' \leq \mathsf{v_{ub}} \end{bmatrix}$$

The continuous evolution of the physical world is expressed by the formula $\mathsf{World}_d(\mathcal{W})$, the details of which we will explain in Section III. Intuitively, it captures the fact that the continuous variables evolve according to the specified differential

equations. We prove this obligation using our adaptation of Platzer's differential induction proof rule [13], which justifies a technique for proving invariants of a system of differential equations without computing an explicit solution. Roughly speaking, differential induction captures the fact that $e_1 \leq e_2$ is preserved by a continuous transition (e.g. $\mathsf{World}_d(\mathcal{W})$) if the derivative of e_1 is less than or equal to the derivative of e_2, under the constraints given by $\mathcal{W}$. Applying differential induction leaves us to prove a first order formula over real arithmetic.

Proving these four goals completes the proof of (1). The most difficult part of the proof was in finding an appropriate inductive invariant. In this case, the arithmetic reasoning was within the scope of Coq's built-in automation.

D. Sensors & Composition

While the velocity shim does guarantee the safety property, it requires an assumption about an input, namely that $v \leq v_{max}$. We could modify the specification of the velocity shim so that it specifies the transition behavior of v_{max} in a way that guarantees that $v \leq v_{max}$, thus removing the assumption. However, this would require reproving the safety theorem of the velocity shim (formula (1)). By leaving the sensor under-specified in the velocity shim, we are able to compose the velocity shim with *any* system that guarantees $\Box v \leq v_{max}$, without needing to reprove the safety theorem of the velocity shim. In this section, we show how to do this for several examples, using a general composition rule for our Sys abstraction.

1) Sensor Error: We start with a simple specification of a sensor that can read the value of v to within some error ϵ. To do so, we first define $\mathsf{Sensor}\big(S, \mathcal{W}, \Delta\big)$, which takes an RTLA state formula S and a real number Δ, and produces an RTLA formula:

$$\mathsf{Sensor}\big(S, \mathcal{W}, \Delta\big) \equiv \mathsf{Sys}\ S\ (\mathcal{W}, S')\ \mathsf{True}\ \Delta$$

This formula expresses the system in which S holds initially, holds after every continuous transition, and all continuous variables in S are unchanged by the discrete transition. Intuitively, S is intended to express the relationship between the physical variable that the sensor is tracking and the actual value it reads. For a sensor of some physical variable x, this relationship is $x - \epsilon \leq x_{sense} \leq x + \epsilon$, where x_{sense} is the sensed value. However, for our purposes, we actually need an upper bound on x, which we accomplish by offsetting x_{sense} by ϵ:

$$\begin{aligned}&\mathsf{Sense}(x, x_{max}) \equiv \\ &\quad x - \epsilon \leq x_{sense} \leq x + \epsilon\ \wedge\ x_{max} = x_{sense} + \epsilon\end{aligned}$$

In order to satisfy the assumption of the velocity shim, we instantiate Sense with v and v_{max} and need to prove that for any $\mathcal{W}$, Δ, and $\epsilon \geq 0$,

$$\vdash \mathsf{Sensor}\big(\mathsf{Sense}(v, v_{max}), \mathcal{W}, \Delta\big) \rightarrow \Box v \leq v_{max} \qquad (2)$$

This theorem follows from SysInd and simple reasoning about linear real arithmetic.

2) Composition: We are now in a position to compose the sensor module with our velocity shim. First, let Sys composition ($\circ$) be defined by conjoining corresponding formulas:

$$\begin{aligned}&\mathsf{Sys}\ I_a\ (\mathcal{W}, C_a)\ D_a\ \Delta \circ \mathsf{Sys}\ I_b\ (\mathcal{W}, C_b)\ D_b\ \Delta \equiv \\ &\quad \mathsf{Sys}\ (I_a \wedge I_b)\ (\mathcal{W}, (C_a \wedge C_b))\ (D_a \wedge D_b)\ \Delta\end{aligned}$$

Note that since all RTLA formulas operate on the same state variables, conjunction is a very general notion of composition.

Using the definition of $\circ$, we can state the theorem that the composition of our sensor with our velocity shim satisfies the safety property $\Box v \leq \mathsf{v_{ub}}$ without any assumptions on v_{max}:

$$\vdash \mathsf{Sensor}\big(\mathsf{Sense}(v, v_{max}), \mathcal{W}, \Delta\big) \circ \mathsf{VelShim} \rightarrow \Box v \leq \mathsf{v_{ub}}$$

This theorem follows immediately from SysCompose (shown in Figure 2). SysCompose states that if the first system guarantees a property, then the second system can assume that property when it proves its safety condition. The combined system does not need the assumption; it has been satisfied by the first system, and has both properties. Similar to SysInd, SysCompose abstracts all of the reasoning for manipulating the internals of the Sys abstraction. Crucially, when we apply SysCompose, we do not need to reprove any theorems about the two systems. Instead we can simply use the soundness proofs of the sensor and velocity shim to satisfy the premises of SysCompose.

3) Delay Compensation: When we compose the sensor specification with the velocity shim, we implicitly assume that the velocity shim can instantaneously read and compute with the value produced by the sensor module, v_{max}. In reality, due to communication or computation time, this may not be the case. For example, suppose that the sensor module actually outputs some value, represented by the variable v_{max_pre} that cannot instantaneously be used in a safety check. The following system compensates for this delay

$$\mathsf{DelayComp} \equiv \mathsf{Sys}\ I\ (\mathcal{W}, \mathsf{True})\ D\ \Delta$$

where $\mathcal{W} \equiv \dot{v} \leq T - g$ as before and

$$\begin{aligned}I &\equiv v_{max} = v + \Delta \cdot \mathsf{max}(0, T - g) \\ D &\equiv {v_{max}}' = v_{max\_pre} + \Delta \cdot \mathsf{max}(0, T' - g)\end{aligned}$$

In this system, D uses the current value of v_{max_pre} to compute an upper bound on v for the next Δ time.

The correctness property for this system is

$$\Box v \leq v_{max\_pre} \vdash \mathsf{DelayComp} \rightarrow \Box v \leq v_{max} \qquad (3)$$

Notice that this property relies on the assumption $\Box v \leq v_{max\_pre}$. However, we can use the sensor module above to satisfy this assumption and use SysCompose to verify the combined system without any assumptions, again without reproving the properties of the individual systems:

$$\begin{aligned}&\vdash \mathsf{Sensor}\big(\mathsf{Sense}(v, v_{max\_pre}), \mathcal{W}, \Delta\big) \circ \mathsf{DelayComp} \\ &\quad \rightarrow \Box v \leq v_{max}\end{aligned}$$

Now we have a new system that guarantees the assumption of the velocity shim, so we can compose them and easily prove the theorem:

$$\begin{aligned}&\vdash \mathsf{Sensor}\big(\mathsf{Sense}(v, v_{max\_pre}), \mathcal{W}, \Delta\big) \circ \mathsf{DelayComp} \circ \mathsf{VelShim} \\ &\quad \rightarrow \Box v \leq \mathsf{v_{ub}}\end{aligned}$$

This approach can be continued for any other sensors, shims, or full-blown controllers that can be specified and verified within the Sys abstraction.

E. Height shim

In addition to controlling velocity through a first-derivative, we have used our deductive approach to control position through a second derivative. In this section we describe our implementation of a shim to enforce an upper bound on height by controlling acceleration. Note that if we were able to directly set the velocity then we could reuse the velocity shim (and its proof) simply by renaming the variables. Since directly setting velocity is unrealistic, we built a new shim that bounds position by setting its second derivative.

$$\mathsf{HeightShim} \equiv \mathsf{Sys}\ I\ (\mathcal{W}, C)\ D\ \Delta$$

where

$$\begin{aligned}
I \equiv\ & \forall t, 0 \le t \le \Delta \rightarrow \\
& (0 \le v + (T{-}g)t \rightarrow \\
& \quad y{+}\mathsf{td}(v, T{-}g, t) + \mathsf{sd}(v{+}(T{-}g)t) \le \mathsf{y_{ub}}) \wedge \\
& (v + (T{-}g)t < 0 \rightarrow y + \mathsf{td}(v, T-g, t) \le \mathsf{y_{ub}}) \\
\mathcal{W} \equiv\ & \dot{y} = v,\ \dot{v} \le T - g \\
C \equiv\ & \mathsf{True} \\
D \equiv\ & \mathsf{td}(v_{max}, a_c, \Delta){+}\mathsf{sd}(v_{max}{+}a_c\Delta){+}y_{max} \le \mathsf{y_{ub}} \\
& \wedge T'{=}T_? \\
\vee\ & T'{=}T_{min}
\end{aligned}$$

$$\mathsf{td}(v, T, \Delta) \equiv v \cdot \Delta + \tfrac{T\Delta^2}{2} \qquad \mathsf{sd}(v) \equiv -\tfrac{v^2}{2\cdot(T_{min}-g)}$$
$$a_c \equiv \max(0, T_? - g) \qquad T_{min} < g$$

The approach is similar to the approach of the velocity shim. Each time this shim runs, it checks whether it will be able to stop in time if it issues the maximum breaking acceleration (T_{min}) the next time the shim runs. The function td computes a conservative upper-bound on the height at the end of Δ time and sd computes the stopping distance assuming T_{min} breaking acceleration. We have formally proven that the height shim guarantees that y never exceeds $\mathsf{y_{ub}}$, under the assumption that y_{max} and v_{max} are bounds on their respective physical variables. Formally,

$$\Box(y \le y_{max} \wedge v \le v_{max}) \vdash \mathsf{HeightShim} \rightarrow \Box y \le \mathsf{y_{ub}}$$

As with the velocity shim, we can compose the height shim with modules guaranteeing the assumptions that the height shim makes on y_{max} and v_{max}. Using SysCompose, we can easily prove that the composed system guarantees $\Box y \le \mathsf{y_{ub}}$ from the individual proofs.

Verifying the height shim differs from the velocity shim in two ways. First, the differential equations describing the physical evolution of the system, the shim logic, and therefore the inductive invariant are all more complex. This in turn means that the real arithmetic proof obligations are substantially more intricate. In practice, this means that the existing foundational, nonlinear real arithmetic decision procedure is not able to solve all of the goals, even though the unverified SMT solver Z3 [14] solves all goals quickly. Second, the verification used history variables (omitted from the specification in this paper for simplicity) to record the value of each physical variable in the last discrete transition. We use these values to describe the safety buffer that the system consumes during the continuous transition. These variables do not change the behavior of the shim in any way; they are used only for reasoning.

Benefits of Composition: When verifying our two shims, the vast majority of the verification effort was devoted to foundationally reasoning about real arithmetic proof obligations. Our composition technique takes a step towards reducing that burden. As a point of comparison with the non-compositional approach, our first implementation of the height shim was monolithic, including all of the code for reasoning about delay compensation (but not sensor error). The result was more complex real arithmetic goals containing larger expressions and more variables. When we verified the height shim compositionally, the arithmetic proof obligations were simpler and, as a result, required less manual proof effort to simplify the goals into a form that the foundational decision procedures could handle. Moreover, verifying the height shim with the noisy sensor was simply a matter of combining the independent proofs using SysCompose. This is a promising result considering that the number of variables influences the complexity of the inductive invariant which is directly related to complexity of automatic verification.

Finding Inductive Invariants: In general, one of the challenges of formal verification lies in building a suitable inductive invariant, and hybrid systems are no exception. However, we have found that developing the inductive invariant is actually a part of the process of developing the shim. For example, when building the velocity shim, we first built the inductive invariant stating that the thrust is safe until the next time the shim runs. We then built a shim to compute a thrust that will be safe until the next time it runs; this followed naturally from the inductive invariant. This means that we have not found the task of finding an inductive invariant to be an *additional* burden on top of the necessary task of building the shim itself. Instead, we found these two tasks to be naturally related, regardless of whether or not one performs foundational verification.

III. TLA

As we have already discussed, our Sys abstraction unfolds to an RTLA formula, which is expressed within Coq. While Sys is designed to model hybrid systems, RTLA is a general purpose formalism that does not have any constructs specific to hybrid systems. In this section, we explain how we define Sys on top of our embedding of RTLA in Coq.

A. Embedding RTLA inside of Coq

To gain the power of a general purpose, foundational proof assistant in addition to the benefits of our TLA-inspired formalism, we developed RTLA as an embedded domain specific *logic* inside of Coq. RTLA primarily differs from TLA in three ways. First, since the majority of our reasoning is numeric and about the physical world, RTLA fixes all variables to take on real number values. Second, we need some mechanism for specifying physical dynamics, which are most naturally described using differential equations. Rather than extending RTLA with a specialized formalism of differential equations, we instead allow embedding arbitrary Coq state relations directly into RTLA. This enriching of RTLA with arbitrary Coq propositions is analogous to Lamport's enriching of TLA with set theory to build TLA+ [15]. Third, RTLA allows formulas that are not invariant under steps where none of the formula's variables change (stuttering). This was not an

issue for our examples, but in the future, we plan to study the importance of stuttering-invariance for other hybrid systems.

B. Sys in TLA

Our Sys abstraction captures the various pieces that compose a hybrid system and expresses them as an RTLA formula. At an intuitive level (and temporarily ignoring timing constraints), Sys must encode the fact that a system can evolve either continuously according to some differential equations or discretely according to the discrete program. Throughout the rest of this section, c refers to the set of continuous variables and d the set of discrete variables – formally $c = \mathit{nonzero}(\mathcal{W}) \cup \mathit{free}(C)$ and $d = \mathit{free}(D)$. The untimed version of Sys is the following:

$$\begin{array}{l}\mathsf{Sys}^{\mathsf{untimed}}\ I\ (\mathcal{W}, C)\ D \equiv \\ \quad I \wedge \Box \left[\begin{array}{lll} & \mathsf{World}_d(\mathcal{W}) \wedge C & \text{(continuous)} \\ \vee & \mathsf{Discr}_c(D) & \text{(discrete)} \end{array} \right]\end{array}$$

$\mathsf{Discr}_c(D)$ describes the discrete evolution of the system using the action formula D. We additionally require that the values of the continuous variables are unchanged.

$$\mathsf{Discr}_c(D) \equiv D \wedge \mathsf{Unchanged}(c)$$

The interesting part of Sys arises when we describe the continuous evolution of the world. The first part of the world transition captures the evolution of the continuous variables over time using the following definition:

$$\begin{array}{l}\mathsf{Continuous}(\dot{x_1} \sim_1 e_1, \ldots, \dot{x_n} \sim_n e_n) \equiv \\ \quad \exists (r : \mathbb{R})\,(f : \mathbb{R} \to \mathsf{Var} \to \mathbb{R}), 0 < r \\ \qquad \wedge \mathsf{Solves}(f, \dot{x_1} \sim_1 e_1, \ldots, \dot{x_n} \sim_n e_n, r) \\ \qquad \wedge x_1 = f(0, x_1) \wedge \ldots \wedge x_n = f(0, x_n) \\ \qquad \wedge {x_1}' = f(r, x_1) \wedge \ldots \wedge {x_n}' = f(r, x_n)\end{array}$$

Here, r is the amount of time that the system evolves for, f is a solution to the differential (in)equations (expressed by the Solves predicate using Coq's real analysis library, with $\sim_i \in \{=, <, \leq, >, \geq\}$). The final two lines relate the starting state to the value of the differential equation at 0 and the final state to the value of the differential equation at time r. We build the dynamics of the world on top of Continuous using the following formula.

$$\begin{array}{l}\mathsf{World}_d(\dot{x_1} \sim_1 e_1, \ldots, \dot{x_n} \sim_n e_n) \equiv \\ \quad \mathsf{Continuous}(\{\dot{x_1} \sim_1 e_1, \ldots, \dot{x_n} \sim_n e_n\} \cup \{\dot{d_i} = 0\})\end{array}$$

It is important to note that setting the derivatives of the discrete variables to 0 is a stronger statement than simply saying that they are unchanged since the derivatives of continuous variables can depend on the discrete variables.

At first glance, our definition of continuous transitions may seem strange, since any single sequence of states satisfying Sys does not capture all intermediate states of the system. Instead, the discrete trace captures finite observations along the continuous evolution of the physical world. In fact, it may seem as though a sequence of states is a poor fit for describing the continuously evolving physical world since time and other continuous variables can only advance in discrete steps. Lamport argues for the adequacy of this encoding in TLA+ [16] and Platzer gives a more detailed treatment of the argument in [13]. The core of the argument lies in the fact that in RTLA (and TLA) we prove properties of *all* sequences of states rather than properties of a single sequence of states. This means that when we prove $\vdash \mathsf{Sys}\ I\ (w, C)\ D\ \Delta \to P$ for some property P, we are proving that P holds on *all* sequences of states that satisfy $\mathsf{Sys}\ I\ (w, C)\ D\ \Delta$. In other words, we are proving properties of all possible sequences of samplings of the hybrid system's state. Even if a particular sequence skips a state in the evolution of the physical world, the definition of Continuous allows another sequence to contain that state.

Expressing Timing Constraints: The primary limitation of $\mathsf{Sys}^{\mathsf{untimed}}$ is that it does not express timing constraints. This deficiency allows the world to execute for as long as it wants, which is clearly undesirable if the discrete system is to enforce non-trivial properties. To express timing constraints, we enrich the $\mathsf{Sys}^{\mathsf{untimed}}$ definition into the definition of Sys by adding the highlighted pieces:

$$\begin{array}{l}\mathsf{Sys}\ I\ (\mathcal{W}, C)\ D\ \Delta \equiv \\ \quad I \wedge 0 \leq \tau \leq \Delta \wedge \Box \left[\begin{array}{l} \mathsf{World}_d(\mathcal{W} \cup \{\dot{\tau} = -1\}) \wedge C \wedge 0 \leq \tau' \\ \vee\, \mathsf{Discr}_c(D) \wedge 0 \leq \tau' \leq \Delta \\ \vee\, \mathsf{Unchanged}(d \cup c \cup \{\tau\}) \end{array} \right]\end{array}$$

The τ variable is a timer that represents the maximum amount of time that the world can run for before the discrete system must run. In the continuous transition, this timer decreases with a derivative of -1 and the constraint $0 \leq \tau'$ prevents the transition from evolving for too long. The second conjunct in the discrete transition allows the discrete transition to pick any amount of time between 0 and Δ. We choose this formulation because it allows us to prove that all properties of slower systems also hold when the system runs faster, i.e. when Δ is smaller.

IV. From Model to Reality

In addition to modeling and verifying our systems, we also implemented both the velocity and height shims to run on a 3D Robotics Iris+. Running the system whose model we verify is important because it allows us to experimentally evaluate the gap between the model and the actual system that we run. We can divide this gap into two pieces: the gap due to our model of the physical world (Section IV-A), and the gap due to our model of the discrete controller (Section IV-B). We conclude by discussing some of the insights that we gained from running our code on an actual quadcopter (Section IV-C).

A. A Small Model in a Big World

The primary gap between the physical world and our model lies in the fact that our model captures only the vertical dimension. In particular, it does not model the orientation of the quadcopter and therefore does not capture the direction of thrust. However, a model that includes attitude is a refinement of the world model that our specifications use. This means that all traces allowed by a model including attitude are also allowed by the world model used in our specifications. As we explained in Section II-B, this is because our specifications model the world using differential *inequalities* stating that the vertical thrust is upper-bounded by the thrust produced by the motors. This relaxation of the specification means that we must prove properties of a more liberal system, but it allows us to use our results in the more constrained, richer model which includes attitude.

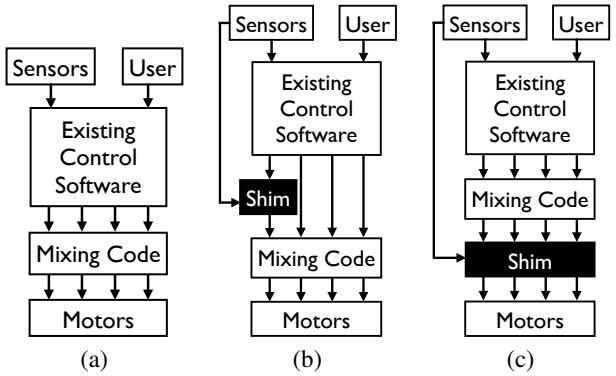

Fig. 3. A simplified depiction of the ArduPilot architecture (a) without any modification, with our shim running (b) before and (c) after the motor mixing code. Black boxes denote verified code.

The other discrepancies between our model and the real world are common simplifying assumptions of models for verification purposes. For example, modeling external factors such as wind, air resistance, etc. would be possible by adding extra terms into the differential world description. Finally, our model also relies on the common assumption of instantaneous change of discrete variables such as thrust. In principle, output values such as thrust actually change over a very small amount of time. The wealth of literature suggests that this is a reasonable assumption but it nonetheless constitutes a formal discrepancy.

B. From Relations to Bits

Our description of the shim architecture in Section II describes the shim as the last piece of code that runs before signals are sent to the motors. This is not strictly necessary. There are advantages and disadvantages to running the shim at different points.

Figure 3.(a) depicts a simplified version of the ArduPilot architecture without any of our modifications. The existing control software takes input from the user and sensors and outputs a desired throttle, roll torque, pitch torque, and yaw torque to the "motor mixer" module. This module then computes the signals to send to each of the four motors to best approximate the desired behavior. The approximation is necessary because it is not always possible to achieve all four desired values simultaneously since the quadcopter relies on differences between motor speeds to induce non-zero torques.

One place to execute our shim is before the motor mixing module (Figure 3.(b)). At this point, our shim executes directly on the desired throttle of the higher-level controller. To meet the interface of our specifications of both the height and velocity shims, we must convert this input into a desired vertical thrust ($T_?$). We must also convert the thrust output by the shims (T) into a desired throttle that serves as the input to the mixer module. We accomplish both tasks by multiplying the throttle by a constant which we determined empirically. We discuss the consequence of this choice in Section IV-C. As we have already discussed, this will actually provide an upper bound on vertical acceleration, an acceptable input to our shims. Placing the shim here allows the motor mixer to optimize the engine outputs to achieve the other parameters (attitude torques) as best it can. However, it also requires us to trust that the motor mixer module never exceeds the desired throttle, a property that we believe to hold but have not formally verified.

Running after the motor mixer (Figure 3.(c)) allows us to remove the mixer from the trusted computing base. To meet the interface at this level, we must translate between the motor signals and the induced thrust. We again accomplish this using an empirically determined constant that we use to scale each of the motor signals. If the shim rejects the proposed thrust, then it must compute new signals for the motors to induce a safe thrust. There are many ways to achieve a particular total thrust by adjusting four motor signals. In order to minimize the affect of the shim on attitude dynamics, we linearly scale back each of the motor values to achieve the thrust output by the shim.

Regardless of where we insert the shim, the trusted computing base still includes the sensor fusion code that runs on the quadcopter. We use this code to provide bounds (v_{max} and y_{max}) on physical variables, in essence treating the sensor fusion code as an unverified sensor module. This code is substantially larger and more complex than the motor mixing code and we are interested in applying our techniques to reason about it in the future. We are also trusting our (currently manual) translation of the model from RTLA to C code. This translation includes picking an appropriate value for Δ, the maximum time between discrete transitions. However, any upper bound suffices since our proofs hold for all Δ. Finally, we ignore the formal gap between real arithmetic used in our models and floating-point arithmetic used in the running code. In future work we plan to close this formal gap using Coq's libary for reasoning about floating point computation [17].

C. Empirical Results

Evaluating empirical results of this nature is important when exploring models. For example, when we first described our shim logic to an expert pilot he was concerned that disengaging the motors so harshly might have a destabilizing effect on the quadcopter. It was only experimentally that we learned that this was not the case.

Experimentally, both the velocity and the height shim enforce their respective safety properties. The height and velocity shims allowed the quadcoptor to go right up to the provided height or velocity bounds. In some rare cases, the quadcopter went above the bounds, by a small amount, for example about ten centimeters for a height bound of 30 meters. We attribute these small violations to un-modeled forces such as wind, sensor inaccuracy, and inaccuracy in the measured relationship between throttle/motor signals and thrust induced. In fact, we found the measured relationship between signals and thrust to have a significant impact on the behavior of the quadcopter and was perhaps the greatest source of error that we noticed. We were careful to be conservative when measuring these constants; since our shims both provide upper bounds, we can safely err on the side of constants that provide upper bounds on acceleration. In the future, we plan to investigate running our shims on top of closed-loop acceleration controllers to avoid the need for these empirical constants.

We flew the quadcopter in both loiter and stabilize modes, and also had the quadcopter approach the height and velocity bounds from a variety of velocities and orientations. In all cases, the shims enforced their safety properties with rare, small violations, and allowed us to retain control over the quadcopter. We never ran both shims simultaneously because we have no verification results for this scenario. Fundamentally, to compose the shims we are obliged to show that the shims do not conflict with one another, e.g. by requiring different remedial actions.

Also, recall that our height shim is conservative in the way that it estimates upward thrust: it assumes that the thrust requested by the higher-level controller would be applied directly in the upward direction, even if the attitude of the quadcoptor is not upward. As a result, if the attitude is not level, our height shim assumes that there is a larger upward thrust than really occurs, and so it will engage earlier than it needs to. We noticed this effect experimentally: when approaching the height through a non-level attitude, the quadcopter would stop ascending at a lower height than the actual bound. Furthermore, when the quadcopter is at the height bound with the shim engaged and the upward throttle stick engaged to the maximum, if we start rolling or pitching, the quadcopter will not only move in the x-y direction, but it will also descend slightly, since as the orientation changes, the height shim becomes more conservative.

Finally, we also tried our velocity shim with a small negative velocity as the bound. With the throttle stick engaged to the maximum, this caused the quadcopter to land while allowing us to control other aspects of the flight such as attitude and x-y positioning.

V. Discussion

In this section we discuss the motivation for our design decisions. We begin by motivating the use of foundational proof assistants (Section V-A) including some anecdotal evidence for the benefits of foundational proofs. Next, we motivate our choice of RTLA as a formalism (Section V-B) highlighting some of the useful principles that underlie its design.

A. Benefits of Foundational Proof Assistants

Proof assistants provide strong guarantees: they require all proofs to be done in full formal detail, with an unparalleled attention to every single detail. This attention to detail can find subtle but critical problems that otherwise might go unnoticed. For example, we initially axiomatized differential induction instead of proving it sound within our framework. We used this axiom to "verify" a version of the height controller that was in fact not safe (note also that the height controller itself is not trivial, and it is actually quite easy to get it wrong). This left us in an unfortunate state of blissful ignorance: we had a subtle bug in our shim, but we did not uncover it at first, *even though we were applying formal methods*, because our statement of differential induction was subtly incorrect. It was only when we attempted to foundationally prove, rather than axiomatize, differential induction that we found our initial phrasing of the proof rule to be unsound. Fixing the statement led us to find several issues with earlier versions of our formalism that appeared reasonable but were, in fact, insufficiently constrained and therefore not correct. These anecdotes underscore the key benefit of foundational proofs: they provide a high level of confidence.

In addition to the foundational guarantees, proof assistants provide very expressive, general purpose logics, which can serve as the foundation of a wide range of interesting theories. We are already leveraging the standard library's real analysis theories to reason about real arithmetic and calculus. As we extend our verification beyond the shims that we have developed, the full power of Coq's rich logic and all of the theories built up in it are at our disposal.

The deductive reasoning style embodied in proof assistants provides useful feedback when verification fails since the user is guiding the proof. When a proof does not work, the user is aware of all of the steps taken in the development and often learns enough from the failed proof to be able to fix the system. This feedback is complementary to counter-examples provided by tools such as Z3 [14], which do not provide foundational proofs for nonlinear real arithmetic. This process of building proofs also makes the developer more aware of the minimal assumptions that a hybrid system is making and the maximal guarantees that it can ensure. In practice, we have found that proofs lead us to find more general, and therefore compositional, interfaces that are easier to satisfy. Further, we believe that the search for these interfaces is the key to richer forms of composition such as conjoining multiple shims that operate on dependent variables such as height and velocity, which we do not address in this work.

Proof assistants also allow us to build automation without sacrificing foundational proofs. In this work, much of that automation surrounds our embedding of temporal logic. This automation is scripted in $\mathcal{L}_{\text{tac}}$, which means that the automation lies completely outside of our trusted computing base. This property has allowed us to rapidly iterate on the automation without needing to worry about its soundness.

Automation for Real arithmetic: The biggest drawback to the use of foundational proof assistants for formalizing hybrid systems is the lack of good automation for reasoning about real arithmetic. For example, the only real arithmetic decision procedure currently implemented for Coq is `psatz` [12] which can be extremely slow, even for simple goals. This is especially true when the goal is unprovable; `psatz` can run for hours before overflowing the stack. While developing our proofs we often passed real arithmetic goals to Z3 to sanity check them before running them through `psatz`. The value of doing this throughout the development process could easily be measured in hours (or days) of productivity gained. Our composition theorems are another step towards reducing the proof burden by factoring the problem into more manageable pieces. We believe that even more abstraction is possible by codifying the "tricks of the trade" as formal, general-purpose proof rules.

B. Benefits of TLA

Now we turn to our choice to model RTLA on TLA and define the Sys formalism on top of it. TLA is only one of many logics that have been proposed for reasoning about hybrid systems [6], [13]. In this section we seek to illuminate some of the benefits of TLA that are inherited by RTLA. We will discuss other logics in Section VI.

One of the biggest benefits to using TLA is that refinement is natural to express; it is simply normal logical implication. For example, the following (standard) proof rule gives us the ability to derive a property about a more concrete system (P) using a proof about a more abstract system (P') given that P implies (is a refinement of) P':

$$\frac{\vdash P \to P' \qquad \vdash P' \to Q}{\vdash P \to Q}\ \textsc{RefineL}$$

Using this rule we can show that the sensor with error from Section II-D can be used to implement the upper bound velocity sensor needed by the velocity shim. TLA also uses refinement for building simple abstractions. For example, rather than thinking of the velocity shim as missing the sensor component, we can instead think of it as being defined with a very under-specified sensor, and the sensor module provides a refinement of that specification. In this way, composition and refinement are one and the same.

Refinement is particularly powerful in TLA because transitions are unconstrained by default. Unlike in a programming language where `x = y + 1` means that `x` changes and the rest of the world stays the same, in TLA, $x' = y + 1$ allows the rest of the world to evolve arbitrarily. This allows other components to execute "in parallel" on their own state simply by joining the programs using conjunction. The only drawback to this approach is that care must be taken to avoid these programs from interfering with one another in unexpected ways. We avoid this problem by enhancing our formalism with a mechanism for renaming variables in RTLA formulas.

VI. Related Work

There has been a tremendous amount of work on the specification and verification of hybrid systems. In this section, we describe some of the prior work in this area, highlighting the commonalities and difference with our own work. In general, the primary difference lies in our use of **foundational**, **deductive** proofs.

Roughly speaking, prior work on specification and verification of hybrid systems falls into two categories: logics for hybrid systems and hybrid automata.

A. Logics for hybrid systems

A number of logics have been developed for reasoning about hybrid systems. In most cases, we distinguish ourselves by completely formalizing our logic within a foundational proof assistant.

Closely related to our development is Platzer's differential dynamic logic (∂DL) [13]. ∂DL includes a deductive proof calculus for reasoning about hybrid systems. A key to reasoning about continuous transitions in ∂DL is differential induction [13]. In this work, we adapted a simplified version of differential induction to our framework and proved it sound in the underlying formal semantics of RTLA. To the best of our knowledge this is the first mechanization of the differential induction proof.

Platzer implemented ∂DL in KeYmaera [13] and has used it to verify a number of interesting problems including: an airborne collision avoidance maneuver [18], intelligent cruise control [19], and a robot for performing surgery [20]. While KeYmaera is deductive, it is not foundational and includes complex external solvers within its trusted computing base [13]. In addition, the generality of our underlying logic allows us to leverage other Coq developments without compromising our small trusted core.

There has also been work on formalizing hybrid systems inside TLA+ [10], [16], [21]. These developments, just like our own, use Lamport's technique for expressing continuous time within the propositional formalism thus avoiding pulling this complexity into the underlying formalism. While TLA+ has been embedded in the Isabelle proof assistant [22], to the best of our knowledge, none of the prior work on hybrid systems has built on this embedding.

Recently, Anand and Knepper performed an impressive deductive Coq verification of properties of ground robots, using an embedding of a hybrid extension of the logic of events [9]. However, they do not provide a proof calculus for their logic, as we do. This proof calculus allows us to easily perform compositional verification, thus reducing the foundational proof burden. Moreover, our choice to verify shims allows us to retain the benefits of complex off-the-shelf control software while enforcing safety guarantees.

Finally, there has been prior work on developing temporal logics specifically for real time and hybrid systems [6], [23]. In these logics, time and other continuous variables are given special treatment and the behavior of these variables appears in the semantics of the logic. In this dimension, our choice of TLA is motivated by the desire to easily reason about refinement—a key property in closing the gap between models and their implementations.

B. Hybrid Automata

Hybrid automata are a popular formalism for modelling hybrid systems [24]. A hybrid automaton can be represented by a graph where a node represents a control mode with an associated continuous evolution (typically specified by a system of differential equations) and edges represent discrete transitions of the system.

Verifying safety properties of a hybrid automaton is typically done by model checkers. There are a number of highly successful implementations such as HyTech [4] and PHAVer [5]. Unfortunately, since hybrid systems are inherently infinite-state, the reachability problem is undecidable in the general case [25]. In fact, reachability is only decidable for a very restricted class of automata, for example rectangular automata. Even small generalizations of these classes lead to undecidability.

To address the problem of infinite states, it is common to verify a finite state abstraction of a hybrid automaton. Many techniques have been developed to automatically compute finite-state abstractions of hybrid automata [26], [27]. However, automatically constructing suitable abstractions is not always successful and manually constructing an appropriate abstraction can be difficult.

One of the benefits of building on top of a general-purpose proof assistant is the ability to use its logical core as a

foundation for composition of reasoning tools. For example, we can use verified model checkers, e.g. the one developed by Niqui [28], to prove properties in a foundational way. Here, the soundness proof of the model checker serves as an abstract proof for all of the properties that it derives. This approach would enable us to leverage previous work on verifying hybrid systems in Coq [28]–[31]. We can then use Coq to prove a refinement relation between the verified automaton and a logical formula in order to use the result within our setting.

Hybrid systems have also been verified in other proof assistants: e.g. PVS [8] and HOL [7]. Beyond the particular choice of proof assistant, our work differs from these by taking a deductive approach and by making heavy use of differential induction to reason about the continuous dynamics.

C. Architectures for hybrid system safety

There has also been work on architecting hybrid systems in order to ensure safety in the presence of complex, unverified controllers. Much of this work has been based on the simplex architecture [11]. In this architecture, there is a simple module that constantly monitors the system and takes control away from more complex modules before the system can enter an unsafe state. We follow a similar principle with our shim architecture.

Livadas and Lynch solve a similar problem using hybrid I/O automata to model and reason about "protectors" for hybrid systems [32]. A protector is designed to ensure a safety property of a particular hybrid system.

VII. Conclusions

In this paper, we formalized a representation of hybrid systems on top of a TLA-inspired formalism inside the Coq proof assistant. We demonstrated how to use our formalization to compositionally build foundational proofs of shims that enforce safety properties of otherwise unmodified systems. We built, verified, and tested shims for enforcing both a maximum vertical velocity and a maximum height. Finally, we reported on our experiences, including some of the difficulty of reasoning about real arithmetic, the benefits of composition when performing this reasoning, the advantages of foundational verification, and some of the benefits of building on top of TLA. In the future, we plan to explore liveness properties of shims and to leverage the expressivity of the underlying Coq formalism to carry our proofs from the high level models all the way down to the bits that run on the quadcopter.

References

[1] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[2] Coq Development Team, "The Coq proof assistant reference manual, version 8.4," 2012.

[3] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294.

[4] T. A. Henzinger, P. Ho, and H. Wong-Toi, "HYTECH: A model checker for hybrid systems," in *CAV '97*, 1997.

[5] G. Frehse, "PHAVer: algorithmic verification of hybrid systems past HyTech," *STTT*, vol. 10, no. 3, pp. 263–279, 2008.

[6] D. Bresolin, "HyLTL: a temporal logic for model checking hybrid systems," *ArXiv e-prints*, Aug. 2013.

[7] N. Vlker, "Towards a HOL framework for the deductive analysis of hybrid control systems," 2000.

[8] E. Abraham-Mumm, U. Hannemann, and M. Steffen, "Verification of hybrid systems: formalization and proof rules in PVS," in *ECCS '01*, 2001, pp. 48–57.

[9] A. Anand and R. Knepper, "ROSCoq: Robots powered by constructive reals," *ITP'15*, vol. 15, p. 2015, 2015.

[10] L. Lamport, "The temporal logic of actions," *TOPLAS*, vol. 16, no. 3, pp. 872–923, 1994.

[11] L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving dependable real-time systems," in *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, vol. 1. IEEE, 1996, pp. 335–346.

[12] F. Besson, "Fast reflexive arithmetic tactics the linear case and beyond," in *TYPES*, ser. LNCS. Springer Berlin Heidelberg, 2007, vol. 4502, pp. 48–62.

[13] A. Platzer, *Logical analysis of hybrid systems: proving theorems for complex dynamics*. Springer Publishing Company, Incorporated, 2010.

[14] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[15] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[16] ——, "Hybrid systems in TLA+," in *Hybrid Systems*, ser. Lecture Notes in Computer Science, R. Grossman, A. Nerode, A. Ravn, and H. Rischel, Eds. Springer Berlin Heidelberg, 1993, vol. 736, pp. 77–102.

[17] S. Boldo and G. Melquiond, "Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, July 2011, pp. 243–252.

[18] A. Platzer and E. M. Clarke, *Formal verification of curved flight collision avoidance maneuvers: A case study*. Springer, 2009.

[19] S. M. Loos, A. Platzer, and L. Nistor, "Adaptive cruise control: Hybrid, distributed, and now formally verified," ser. FM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 42–56.

[20] Y. Kouskoulas, D. Renshaw, A. Platzer, and P. Kazanzides, "Certifying the safe design of a virtual fixture control algorithm for a surgical robot," ser. HSCC '13. New York, NY, USA: ACM, 2013, pp. 263–272.

[21] L. Lamport, "Real time is really simple," MSR-TR-2005-30, Microsoft Research, Tech. Rep., 2005.

[22] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, "Verifying safety properties with the TLA+ proof system," in *IJCAR 2010*, 2010, pp. 142–148.

[23] J. S. Ostroff, *Temporal logic for real-time systems*. Cambridge Univ Press, 1989, vol. 40.

[24] T. A. Henzinger, "The theory of hybrid automata," in *LICS '96*, 1996, pp. 278–292.

[25] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *J. Comput. Syst. Sci.*, vol. 57, no. 1, pp. 94–124, 1998.

[26] 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.*, vol. 14, no. 4, pp. 583–604, 2003.

[27] R. Alur, T. Dang, and F. Ivancic, "Counter-example guided predicate abstraction of hybrid systems," in *TACAS '03*, 2003, pp. 208–223.

[28] M. Niqui and O. Tveretina, "Modular Development of Hybrid Systems for Verification in Coq," in *HSCC '08*, ser. HSCC '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 638–641.

[29] H. Geuvers, A. Koprowski, D. Synek, and E. van der Weegen, "Automated machine-checked hybrid system safety proofs," in *ITP '10*, 2010, pp. 259–274.

[30] P. Collins, M. Niqui, and N. Revol, "A Taylor Function Calculus for Hybrid System Analysis: Validation in Coq (Extended Abstract)," 2010.

[31] O. Tveretina, "Towards the Safety Verification of Real-Time Systems with the Coq Proof Assistant," *JAMRIS*, vol. 3, pp. 30–32, 2009.

[32] C. Livadas and N. A. Lynch, "Formal verification of safety-critical hybrid systems," in *HSCC '98*, 1998.