# Verifying Cyber-Physical Systems by Combining Software Model Checking with Hybrid Systems Reachability*

Stanley Bak
stanleybak@gmail.com

Sagar Chaki
chaki@sei.cmu.edu

## ABSTRACT

Cyber-physical systems (CPS) span the communication, computation and control domains. Creating a single, complete, and detailed model of a CPS is not only difficult, but, in terms of verification, probably not useful; current verification algorithms are likely intractable for such all-encompassing models. However, specific CPS domains have specialized formal reasoning methods that can successfully analyze certain aspects of the integrated system. To prove overall system correctness, however, care must be taken to ensure the interfaces of the proofs are consistent and leave no gaps, which can be difficult since they may use different model types and describe different aspects of the CPS.

This work proposes a bridge between two important verification methods, software model checking and hybrid systems reachability. A *contract automaton* (CA) expresses both (1) the restrictions on the interactions between the application and the controller, and (2) the desired system invariants. A sound assume-guarantee style compositional proof rule decomposes the verification into two parts – one verifies the application against the CA using software model checking, and another verifies the controller against the CA using hybrid systems reachability analysis. In this way, the proposed method avoids state-space explosion due to the composition of discrete (application) and continuous (controller) behavior, and can leverage verification tools specialized for each domain. The power of the approach is demonstrated by verifying collision avoidance using models of a distributed group of communicating quadcopters, where the provided models are software code and continuous 2-d quadcopter dynamics.

## CCS Concepts

**•Software and its engineering → Formal software verification;**

*This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0003464 (SEI); 88ABW-2016-2806, 06 JUN 2016 (AFRL).



*EMSOFT'16, October 01-07 2016, Pittsburgh, PA, USA*



ACM ISBN 978-1-4503-4485-2/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2968478.2968490

## Keywords

verification; cyber-physical systems; compositionality; assume-guarantee; hybrid systems; software model checking.

## 1. INTRODUCTION

A cyber-physical system (CPS) consists of a tight coupling between software and the physical world. CPSs play a crucial role in many aspects of our day-to-day lives, ranging from thermostats, cars and airplanes, to medical devices, nuclear power plants and electric grids. Since many CPSs are safety-critical applications, it is important to assure their correct behavior to the maximum extent possible. Formal verification provides a high level of confidence in a system's operation, and is therefore a desirable assurance approach. However, scalable formal verification of CPSs is an open challenge, and the topic of this paper.

A general CPS may consist of a distributed set of agents in a shared physical environment. Communication is performed over a network, which may not necessarily be reliable. The agents are each implemented using C-language source code that is run periodically by a real-time scheduler. The goal of this work is enable the verification of *high-level properties*, which deal with the physical world and in relation to multiple agents. With this goal in mind, the specific contribution is a decomposition of one part of the larger verification process. In particular, the proposed method enables formal reasoning between the software code on a single agent and the physical environment with which it interacts. Combined with other verification approaches, we show how this decomposition enables end-to-end reasoning about high-level system properties.

We consider a CPS agent consisting of two layers – an application $A$ and a controller $C$. The single-agent system $S$, is a composition of these two, $S = A \parallel C$. This composition is performed along the analysis boundaries, where $A$ is analyzed using software model checking and $C$ is analyzed with hybrid systems reachability tools. Note that in our approach, the controller model $C$ consists of not just the low-level controller, but also the continuous plant dynamics. The application $A$ and controller $C$ execute in parallel and communicate via shared variables. The application is available as source code that calls a specific set of functions (API) to access (read/write) the shared variables, while the plant/controller model is represented by a hybrid automaton which interacts with the environment based on the shared variables.

In order to enable end-to-end reasoning, it becomes necessary to verify *cyber-physical properties*, which are true for the combined application and controller system, but not necessarily for the individual parts. We want to verify that the system satisfies some cyber-physical safety property $\Phi$, i.e.,

$A \parallel C \models \Phi$. There are two challenges to doing this directly. First, there are no algorithms to verify a composition of source code with hybrid automata. Second, even if we developed such an algorithm, using it would likely suffer from state-space explosion for any practical systems, as it would require us to construct $A \parallel C$.

We propose an approach that sidesteps these two problems. Our key intuition is that $A \parallel C \models \Phi$ typically holds because $A$ and $C$ interact in restricted ways, and each "assumes" a specific behavioral pattern that the other "guarantees". In particular, $A$ calls the API functions in a well-defined sequence with parameters that respect certain preconditions, while $C$ maintains invariants involving the physical system state in relation with the shared variables. We leverage this intuition by using a "contract automaton" (CA) $M$ to capture the restricted interaction between $A$ and $C$, as well as the target safety property $\Phi$. Further the CA leads to an "assume-guarantee" style proof rule that enables compositional verification:

$$\frac{A \preceq M \quad C \preceq M}{A \parallel C \preceq M}$$

where $X \preceq Y$ informally means that $X$ "refines" $Y$, i.e., behaviors of $X$ can conform to the behavior of $Y$. We define the premises and conclusion of the proof rule formally, and prove its soundness. We also describe procedures for discharging the premises using domain-specific tools. Specifically, $A \preceq M$ is verified with a software model checker, while $C \preceq M$ is verified using a hybrid systems reachability tool. Finally, we should how the conclusion of the proof rules implies that the invariants of $M$ are transferred to the system $A \parallel C$, thereby verifying that $A \parallel C$ satisfies the target cyber-physical property $\Phi$.

Our approach not only avoids composing $A$ and $C$ (thus ameliorating state-space explosion) but also uses software verification and hybrid verification tools synergistically. Discharging the two premises is still complex, but they build on software model checking and hybrid systems analysis tools, where a lot of progress is being made. Once a cyber-physical property $\Phi$ is proven about individual agents, other verification methods can use $\Phi$ to reason about high-level system properties dealing with the distributed interactions. To the best of our knowledge, this is the first end-to-end formal verification of high-level properties of a distributed cyber-physical system, that includes both the application software and the controller, using a sound combination of software model checking and hybrid systems reachability analysis.

The rest of the paper is organized as follows. Section 2 first presents a running example of an end-to-end verification problem dealing with distributed collision avoidance for quadcopters. Then, Section 3 introduces contract automata, and the compositional proof rule, and applies this to the quadcopter system. Next, Section 4 shows how the premises of our proof rule can be discharged using software and hybrid systems verification tools. In Section 5, we use the developed method to prove cyber-physical properties, and uses these properties to complete the high-level, distributed collision avoidance proof. Finally, we discuss related work in Section 6, followed by a conclusion.

## 2. QUADCOPTER CPS EXAMPLE

We demonstrate our approach by proving end-to-end collision avoidance, in terms of physical distances, between a group of communicating quadcopters moving in a continuous 2-d space. Each quadcopter contains a path-planning application, written in C, that communicates with other quadcopters and updates its own setpoints. The evolution of the quadcopters physical aspects is governed by differential equations, which are not coupled between different quadcopters (although they move in a shared environment). A low-level controller periodically actuates each system based on the current setpoint, according to some real-time scheduling policy like Rate Monotonic [36]. Communication between quadcopters is unreliable, i.e., message delivery is not guaranteed. We assume a finite 2-d space with a fixed number of quadcopters. We also have control over the initial states. The desired high-level property is that no two quadcopters ever collide, i.e., assuming a given quadcopter radius, the distance between any two quadcopters is always greater than $2 * \texttt{HELI\_RADIUS}$.

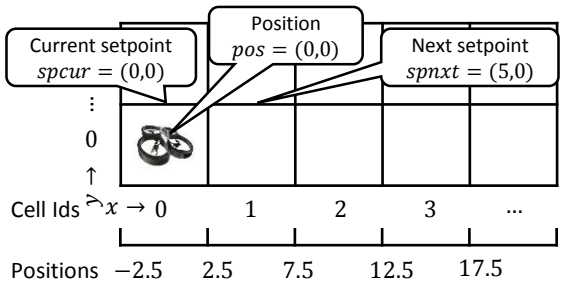



**Figure 1: Example quadcopter.**

The setpoint-generating application needs a strategy to prevent collisions. In our case, the application software generates setpoints based on a 2-dimensional grid which covers the physical space. Each cell is a $5 \times 5$ square, and is represented by a unique pair of integers, which we refer to as the grid id. The left-bottom-most cell has id $(0, 0)$, and the integer ids increase moving to the right or up, as in a traditional coordinate axis. While the application software generates the next setpoint, a controller is responsible for actually moving the quadcopter. The setpoints for each direction are a 5-multiple of a cell id (i.e., they correspond to the centers of cells). The application and controller communicate via two shared variables: (i) $spcur = (spcur_x, spcur_y)$: the current setpoint; and (ii) $spnxt = (spnxt_x, spnxt_y)$: the next setpoint.

In addition, the controller model maintains a variable $pos = (pos_x, pos_y)$ to denote the position of the center of the quadcopter w.r.t. the global coordinate system. For example, in Figure 1, we have $spcur = pos = (0, 0)$ and $spnxt = (5, 0)$, meaning that the center of the quadcopter coincides with the center of cell $(0, 0)$. Given two coordinates $c = (x, y)$ and $c' = (x', y')$ we write $c - c'$ to mean $(x - x', y - y')$, $|c|$ to mean $(|x|, |y|)$ and $c \leq c'$ to mean $(x \leq x' \wedge y \leq y')$. Suppose that the quadcopter is hovering over its current setpoint. Once the application has computed the next setpoint, it moves the quadcopter to this setpoint as follows:

**(Step 1)** The application calls the function $update\_setpoint(x, y)$ where the arguments contains the value of the next setpoint; this function sets the value of $spnxt$ to $(x, y)$, which triggers the controller to move to $spnxt$. Function $update\_setpoint(x, y)$ returns a void value.

**(Step 2)** The application then repeatedly polls the state of the system by calling the function *has_arrived*(), until it returns TRUE. This function returns TRUE only if: $|pos - spnxt| \leq (0.1, 0.1)$. In addition, if *has_arrived*() returns TRUE, it updates the value of *spcur* to be *spnxt*.

Correct interaction between the application and the controller, which we will show is sufficient to prove cyber-physical properties about the combined system, requires several conditions:

**(C1)** The application always calls $update\_setpoint(x, y)$, with arguments that satisfy the condition $|(x, y) - spcur| = (5, 0) \vee |(x, y) - spcur| = (0, 5)$.

**(C2)** Once the application calls $update\_setpoint(x, y)$, it can keep calling *has_arrived*() until it gets a return value of TRUE; once *has_arrived*() returns TRUE, the application can only then start to call $update\_setpoint(x, y)$ again.

**(C3)** When the quadcopter is hovering (i.e., $spnxt = spcur$), the controller must maintain the following invariant: $\Phi_{hover} \equiv |pos - spcur| \leq (1.5, 1.5)$.

**(C4)** When the quadcopter is moving (i.e., $|spnxt - spcur| = (5, 0) \vee |spnxt - spcur| = (0, 5)$), the controller must maintain the following invariant:

$$\begin{aligned} \Phi_{move} \equiv\ & min(spcur_x, spnxt_x) - 1.5 \leq pos_x \\ & \leq max(spcur_x, spnxt_x) + 1.5 \\ \wedge\ & min(spcur_y, spnxt_y) - 1.5 \leq pos_y \\ & \leq max(spcur_y, spnxt_y) + 1.5 \end{aligned}$$

Note that conditions **C1**–**C2** restrict the sequence of function calls that can be made by the application, and the arguments that can be passed, while conditions **C3**–**C4** restrict the behavior of the controller. In the next section, we will see how a contract automaton can be used to both specify and verify such conditions formally.

# 3. CONTRACT AUTOMATON

We assume a computational model where the application and controller execute in parallel, and communicate via three types of shared variables. These shared variables fall in three categories: (i) Cyber variables $V_C$ : these are written by the application only, during function calls; (ii) Parameter variables $V_{Par}$ : these are used as parameters of functions called by the application to interact with the controller; and (iii) Physical variables $V_P$ : these are modified by the controller only. We write $\mathcal{V}$ to denote the set of all variables, i.e., $\mathcal{V} = V_C \cup V_{Par} \cup V_P$. All variables are typed. We use real ($\mathbb{R}$) and Boolean ($\mathbb{B}$) variables. For brevity, we use the symbol for a type to also denote the set of elements of that type. Thus, $\mathbb{R}$ is also the set of all real numbers. Functions can also return void ($\diamond$) values. In our example from Figure 1, we have $V_P = \{pos : (\mathbb{R}, \mathbb{R})\}$, $V_C = \{spcur : (\mathbb{R}, \mathbb{R}), spnxt : (\mathbb{R}, \mathbb{R})\}$, and $V_{Par} = \{x : \mathbb{R}, y : \mathbb{R}\}$.

*Expressions.* Let $\mathbb{D} = \mathbb{R} \cup \mathbb{B}$ be the set of all non-void values (reals, TRUE, and FALSE). Given a set of variable $V \subseteq \mathcal{V}$, we write $Expr(V)$ to denote set of expressions constructed from $V \cup \mathbb{D}$, using numeric operators ($+, -, *, /$, etc.), relational operators ($<, \leq, >, \geq$, etc.), and logical operators ($\wedge, \vee, \neg$, etc.).

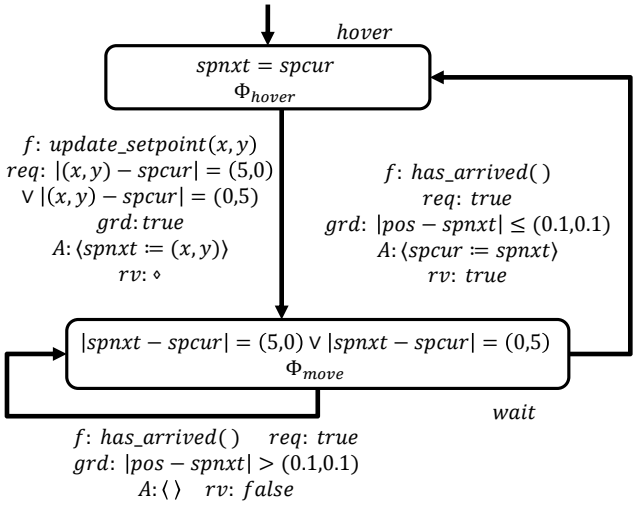



**Figure 2: Example contract automaton.**

*Functions and Function Calls.* A function is a triple $(fn, p, rt)$ where $fn$ is the function name, $p \subseteq V_{Par}$ is a list of its parameters, and $rt$ is its return type. The set of all functions via which the application interacts with the controller is denoted *Func*. Indeed, for our purposes, the semantics of the application is a set of execution traces, where each trace is a sequence of function calls. A function call is a triple $(f, a, rv)$ where $f = (fn, p, rt)$ is a function, $a : p \mapsto \mathbb{D}$ maps each parameter to an argument of appropriate type, and $rv \in rt$ is a return value of appropriate type.

*Example.* In our example from Figure 1, we have $Func = \{f_1, f_2\}$ where: (i) $f_1 = (update\_setpoint, \langle x, y \rangle, \diamond)$; and (ii) $f_2 = (has\_arrived, \langle\rangle, \mathbb{B})$. Some possible function calls are $(f_1, \langle 1, 1 \rangle, \diamond)$, $(f_2, \langle\rangle,$ FALSE$)$, etc.

DEFINITION 1 (APPLICATION). *An application is defined by a C-language program that makes calls to Func.*

DEFINITION 2 (CONTROLLER). *A controller is defined by a hybrid automaton over the variables $V_P \cup V_C$.*

*Assignments.* An assignment is a pair $(lhs, rhs)$ where $lhs \in \mathcal{V}$ is the left-hand side and $rhs \in Expr(\mathcal{V})$ is the right-hand side. The set of all assignments is $Asgn$.

DEFINITION 3 (CONTRACT AUTOMATON). *Formally, a contract automaton (CA) is a 5-tuple $(S, I, T, Inv, L)$ where:*

- *$S$ is a finite set of locations;*
- *$I \in S$ is the initial location;*
- *$T \subseteq S \times S$ is a transition relation;*
- *$Inv : S \mapsto Expr(V_P \cup V_C)$ maps each location to an expression over the physical and cyber variables; informally, $Inv(l)$ is the invariant that a correct controller should maintain when the system is in location $l$;*
- *$L : T \mapsto Func \times Expr \times Expr \times Asgn^* \times (\mathbb{D} \cup \{\diamond\})$ labels each transition with information about the function call from the application that triggers the transition, a guard under which the transition occurs, a sequence of assignments that the transition executes,*

*and the return value of the triggering function call that the transition results in. Formally, if* $L(l, l') = (f, req, grd, U, rv)$*, then it means:*

- *The transition from* $l$ *to* $l'$ *is triggered by a call to function* $f = (fn, p, rt)$ *by the application.*
- *Any such call must satisfy the condition* $req$*, which is an expression over* $V_C \cup p$*.*
- *Once the transition is triggered, it can only occur if condition* $grd$*, which is an expression over* $V_P \cup V_C \cup p$*, holds. The key difference between* $req$ *and* $grd$ *is that while every call to* $f$ *by* $A$ *must satisfy* $req$*, it does not have to satisfy* $grd$*.*
- *If the transition occurs, it executes the assignments in* $U$ *and then the call to* $f$ *returns with value* $rv$*.*

*Note that the labeling of a transition provides a semantic description of the correct implementation of* $f$*. Indeed,* $f$ *must implement the function:*

$$if\ (grd)\ then\ \{U;\ return\ rv;\}$$

*We will use this intuition for the verification steps presented in the following sections.*

*Example.* Figure 2 shows the contract automaton $M$ for the quadcopter system described in Section 2. The automaton has two locations – *hover* and *wait*. The initial location is *hover*. Locations are labeled with corresponding invariants, and transitions are labeled with details about the function calls that trigger them. Note how $M$ enforces the conditions **C1**–**C4** from Section 2. Specifically, conditions **C1**–**C2** are enforced by the possible transitions and the function calls labeling them, while conditions **C3**–**C4** are enforced by the invariants labeling the locations.

## 3.1 Contract Automaton Semantics

To define the semantics of a contract automaton, we have to first define states, and how expressions are evaluated. A state $\sigma : \mathcal{V} \hookrightarrow \mathbb{D}$ is a partial assignment of variables to values. The domain of $\sigma$ is denoted $Dom(\sigma)$. We write $\sigma_1 \oplus \sigma_2$ to denote the state obtained by merging $\sigma_1$ and $\sigma_2$ with disjoint domains, i.e., if $Dom(\sigma_1) \cap Dom(\sigma_2) = \emptyset$, then:

$$(\sigma_1 \oplus \sigma_2)(v) = \sigma_i(v) \,.\, v \in Dom(\sigma_i), i \in \{1, 2\}$$

Given a set of variables $V$, the set of all states $\sigma$ such that $Dom(\sigma) = V$ is denoted $\Sigma(V)$, i.e.,

$$\Sigma(V) = \{\sigma : \mathcal{V} \hookrightarrow \mathbb{D} \mid Dom(\sigma) = V\}$$

Given a state $\sigma$ and a set of variables $V \subseteq Dom(\sigma)$, the projection of $\sigma$ on $V$, denoted $\sigma \downharpoonright V$, is the state such that:

$$Dom(\sigma \downharpoonright V) = V \wedge \forall v \in V \,.\, (\sigma \downharpoonright V)(v) = \sigma(v)$$

Given a state $\sigma$, and an expression $e$, we write $[\![e, \sigma]\!]$ to denote the value obtained by evaluating $e$ under $\sigma$ in the natural way. For example, if $\sigma(v_1) = 5$ and $\sigma(v_2) = 3$, then $[\![v_1 - v_2, \sigma]\!] = 2$, and $[\![v_1 - v_2 > 3, \sigma]\!] = \text{FALSE}$. We write $\sigma \models e$ to mean $[\![e, \sigma]\!] = \text{TRUE}$, and $\sigma \not\models e$ to mean $[\![e, \sigma]\!] = \text{FALSE}$.

*Trajectory.* Given two states $\sigma$ and $\sigma'$ such that $(\sigma \downharpoonright V_C) = (\sigma' \downharpoonright V_C)$, a trajectory $\tau$ from $\sigma$ to $\sigma'$ is the sequence of states encountered as a finite amount of time elapses, due to the continuous dynamics and the low-level controller. This is a trajectory in the hybrid automaton sense, which includes intervals of continuous evolution and discrete jumps. It is an infinite sequence of states starting with $\sigma$ and ending with $\sigma'$ that does not modify the cyber variables, $(\forall \sigma'' \in \tau \,.\, \sigma'' \downharpoonright V_C) = (\sigma \downharpoonright V_C)$. Given expression $e$, we write $\tau \models e$ to mean $\forall \sigma \in \tau \,.\, \sigma \models e$.

*Contract Automaton Transition.* Let $M = (S, I, T, Inv, L)$ be a contract automaton. Its semantics is given by a state transition system, where each state is a pair $(l, \sigma)$ such that $l \in S$ and $\sigma \in \Sigma(V_P \cup V_C)$. There are two types of transitions – application-triggered and controller triggered. An *application-triggered* transition is of the form $(l, \sigma) \xrightarrow{c} (l', \sigma')$ such that: (i) $(l, l') \in T$; (ii) $(\sigma \downharpoonright V_P) = (\sigma' \downharpoonright V_P)$; note that this means an application-triggered transition does not alter the values of physical variables; (iii) $\sigma \models Inv(l) \wedge \sigma' \models Inv(l')$; and (iv) $L(l, l') = (f, req, grd, U, rv)$ and $c = (f, a, rv)$ such that:

– $\sigma \oplus a \models req \wedge grd$; and

– $\{\sigma \oplus a\} U \{\sigma' \oplus a\}$, i.e., state $\sigma' \oplus a$ is obtained from $\sigma \oplus a$ by executing the assignments in $U$; note that we need $a$ since it may be read (but not updated) by $U$.

The set of all application-triggered transitions is denoted $\delta(M)_A$. A *controller-triggered* transition is of the form $(l, \sigma) \xrightarrow{\tau} (l, \sigma')$ such that: (i) $\tau$ is a trajectory from $\sigma$ to $\sigma'$; (ii) $(\sigma \downharpoonright V_C) = (\sigma' \downharpoonright V_C)$; and (iii) $\tau \models Inv(l)$; note this means that the invariant of $l$ is maintained at all intermediate states as $M$ transitions from $\sigma$ to $\sigma'$. A controller-triggered transition does not alter the location of the contract automaton. The set of all controller-triggered transitions is denoted $\delta(M)_C$.

Definition 4 (Contract Automaton Semantics). *An execution of* $M$ *is an alternating sequence of controller-triggered and application-triggered transitions:*

$$(l_1, \sigma_1) \xrightarrow{\tau_1} (l_1, \sigma_1') \xrightarrow{c_1} (l_2, \sigma_2) \ldots (l_{n-1}, \sigma_{n-1}') \xrightarrow{c_{n-1}} (l_n, \sigma_n)$$

*such that:*

$$l_1 = I \wedge$$
$$\forall i \in [1, n-1] \,.\, (l_i, \sigma_i) \xrightarrow{\tau_i} (l_i, \sigma_i') \in \delta(M)_C \wedge$$
$$\forall i \in [1, n-1] \,.\, (l_i, \sigma_i') \xrightarrow{c_i} (l_{i+1}, \sigma_{i+1}) \in \delta(M)_A$$

*The semantics of a contract automaton* $M$*, denoted* $[\![M]\!]$*, is the set of all its executions.*

## 3.2 Refinement

Our broad goal is to show that, if an application $A$ and a controller $C$ both "refine" a contract automaton $M$, then the system composed of $A$ and $C$ refines $M$ as well. In this section, we present this formally. We begin with the semantics of an application.

Definition 5 (Application Semantics). *For our purposes, an application is a black-box that makes calls to functions in Func. Thus, the semantics of* $A$*, denoted* $[\![A]\!]$ *is a set of executions, where each execution* $\pi$ *is a sequence of states and function calls, i.e.,* $\pi = \sigma_1 \xrightarrow{c_1} \sigma_2 \ldots \sigma_{n-1} \xrightarrow{c_{n-1}} \sigma_n$ *such that each* $\sigma_i$ *maps cyber variables to values, i.e.,* $\forall i \geq 1 \,.\, Dom(\sigma_i) = V_C$*.*

Definition 6 (Controller Semantics). *Since the controller* $C$ *is defined by a hybrid automaton, its semantics is given by a set of executions over* $V_C \cup V_P$*, and an initial*

*state $Init_C \in Expr(V_C \cup V_P)$. An execution of $C$ is a sequence $\sigma_1 \xrightarrow{\tau_1} \sigma'_1, \sigma_2 \xrightarrow{\tau_2} \sigma'_2, \ldots, \sigma_{n-1} \xrightarrow{\tau_{n-1}} \sigma'_{n-1}$ such that: (i) $\sigma_1 \models Init_C$; (ii) $\forall i \in [1, n-1] \centerdot (\sigma_i \downharpoonright V_C) = (\sigma'_i \downharpoonright V_C)$; (iii) $\forall i \in [1, n-1]$, $\tau_i$ is a (hybrid) trajectory from $\sigma_i$ to $\sigma_{i+1}$; and (iv) $\forall i \in [1, n-2] \centerdot \sigma_i \downharpoonright V_P = \sigma_{i+1} \downharpoonright V_P$. Intuitively, each trajectory represents evolution of $C$ without interference from $A$, and the possible jump from one trajectory to the next is caused by a function call. The semantics of $C$, denoted $[\![C]\!]$, is the set of all its executions.*

Definition 7 (System Semantics). *The system $S = A \parallel C$ is an asynchronous interleaving of the application and the controller where function calls by the application interleave with the evolution by the controller. The semantics of $S$, denoted $[\![S]\!]$, is given by a set of executions where each execution is a sequence of the form:*

$$\sigma_1 \xrightarrow{\tau_1} \sigma'_1 \xrightarrow{c_1} \sigma_2 \ldots \sigma_{n-1} \xrightarrow{\tau_{n-1}} \sigma'_{n-1} \xrightarrow{c_{n-1}} \sigma_n$$

*such that each transition $\sigma_i \xrightarrow{\tau_i} \sigma'_i$ represents continuous evolution by the controller, and each $\sigma'_i \xrightarrow{c_i} \sigma_{i+1}$ represents a function call by the application. In other words:*

***(SS1)*** $\forall i \geq 1 \centerdot Dom(\sigma_i) = Dom(\sigma'_i) = V_C \cup V_P$

***(SS2)*** $\forall i \geq 1 \centerdot (\sigma_i \downharpoonright V_C = \sigma'_i \downharpoonright V_C) \wedge (\sigma'_i \downharpoonright V_P = \sigma_{i+1} \downharpoonright V_P)$

***(SS3)*** $\sigma_1 \xrightarrow{\tau_1} \sigma'_1, \sigma_2 \xrightarrow{\tau_2} \sigma'_2, \ldots, \sigma_{n-1} \xrightarrow{\tau_{n-1}} \sigma'_{n-1} \in [\![C]\!]$

***(SS4)*** $\sigma_1 \downharpoonright V_C \xrightarrow{c_1} \sigma_2 \downharpoonright V_C \xrightarrow{c_2} \ldots \xrightarrow{c_{n-1}} \sigma_n \downharpoonright V_C \in [\![A]\!]$

Definition 8 (Application Refinement). *$A$ refines $M$, denoted $A \preceq M$, if every execution of $A$, that maintains the invariants in each mode of the contract automaton $M$, corresponds to some execution of $M$. Formally:*

$$\begin{gathered} A \preceq M \iff \\ \forall \sigma_1 \xrightarrow{c_1} \sigma_2 \ldots \sigma_{n-1} \xrightarrow{c_{n-1}} \sigma_n \in [\![A]\!] \centerdot \\ \forall l_1 = I, l_2, \ldots, l_{n-1} \centerdot \\ \forall \tilde{\sigma}_1, \ldots, \tilde{\sigma}_{n-1} \centerdot \sigma_1 \oplus \tilde{\sigma}_1 \models Inv(I) \wedge \\ \forall i \in [2, n-1] \centerdot \sigma_i \oplus \tilde{\sigma}_{i-1} \models Inv(l_i) \wedge \sigma_i \oplus \tilde{\sigma}_i \models Inv(l_i) \implies \\ \exists l_n \centerdot (l_{n-1}, \sigma_{n-1} \oplus \tilde{\sigma}_{n-1}) \xrightarrow{c_{n-1}} (l_n, \sigma_n \oplus \tilde{\sigma}_{n-1}) \in \delta(M)_A \end{gathered}$$

*Note that each $l_i$ is a location of $M$ and each $\tilde{\sigma}_i \in \Sigma(V_P)$ maps $V_P$ to values.*

Definition 9 (Controller Refinement). *$C$ refines $M$, denoted $C \preceq M$, if every trajectory in $C$, that obeys the transitions (ordering and pre/post conditions) from $M$, corresponds to some execution of $M$. Formally:*

$$\begin{gathered} C \preceq M \iff \\ \forall \sigma_1 \xrightarrow{\tau_1} \sigma'_1, \sigma_2 \xrightarrow{\tau_2} \sigma'_2, \ldots, \sigma_{n-1} \xrightarrow{\tau_{n-1}} \sigma'_{n-1} \in [\![C]\!] \centerdot \\ \tau_1 \models Inv(I) \wedge \forall l_1 = I, l_2, \ldots, l_{n-1} \centerdot \\ \forall i \in [1, n-2] \centerdot \exists c_i \centerdot (l_i, \sigma'_i) \xrightarrow{c_i} (l_{i+1}, \sigma_{i+1}) \in \delta(M)_A \implies \\ \tau_{n-1} \models Inv(l_{n-1}) \end{gathered}$$

*This consists of checking that the controller's reachable set of states, under any application $A$ which satisfies the discrete transition conditions in the contract automaton, does not violate the mode invariants in the contract automaton.*

Definition 10 (System Refinement). *$S = A \parallel C$ refines $M$, denoted $S \preceq M$, if every execution of $S$ corresponds to an execution of $M$. Formally:*

$$\begin{gathered} S \preceq M \iff \\ \forall \sigma_1 \xrightarrow{\tau_1} \sigma'_1 \xrightarrow{c_1} \sigma_2 \ldots \sigma_{n-1} \xrightarrow{\tau_{n-1}} \sigma'_{n-1} \xrightarrow{c_{n-1}} \sigma_n \in [\![S]\!] \centerdot \\ \exists l_2, l_3, \ldots, l_n \centerdot (I, \sigma_1) \xrightarrow{\tau_1} (I, \sigma'_1) \xrightarrow{c_1} (l_2, \sigma_2) \\ \ldots (l_{n-1}, \sigma_{n-1}) \xrightarrow{\tau_{n-1}} (l_{n-1}, \sigma'_{n-1}) \xrightarrow{c_{n-1}} (l_n, \sigma_n) \in [\![M]\!] \end{gathered}$$

## 3.3 Cyber-Physical Properties

A contract automaton's power is in proving *cyber-physical properties*. These are properties which are true not solely on the basis of the application software, or the controller, but instead require both to satisfy certain properties (expressed collectively in the contract automaton). The CPS properties are expressed as relations over the cyber and physical variables. For the contract automaton from Figure 2, the corresponding cyber-physical property $\Phi$ is:

$$\begin{gathered} (\Phi_{hover} \wedge spnxt = spcur) \vee (\Phi_{move} \wedge \\ (|spnxt - spcur| = (5, 0) \vee |spnxt - spcur| = (0, 5))) \end{gathered}$$

It follows from Definition 4 that all reachable states of $M$ satisfy its invariant. In other words:

Proposition 1 (Invariant Satisfaction).

$$\begin{gathered} \forall (l_1, \sigma_1) \xrightarrow{\tau_1} (l_1, \sigma'_1) \xrightarrow{c_1} (l_2, \sigma_2) \ldots (l_{n-1}, \sigma_{n-1}) \\ \xrightarrow{\tau_{n-1}} (l_{n-1}, \sigma'_{n-1}) \xrightarrow{c_{n-1}} (l_n, \sigma_n) \in [\![M]\!] \centerdot \\ \forall i \in [1, n] \centerdot \sigma_i \models Inv(M) \wedge \forall i \in [1, n-1] \centerdot \tau_i \models Inv(M) \end{gathered}$$

The main power of Definition 10 is that it implies that if $S \preceq M$, then $Inv(M)$ also holds on $S$ (where $Inv(M)$ is defined as the disjunction of invariants over all locations in $M$ given in definition 3). Formally:

Proposition 2 (Invariant Preservation).

$$\begin{gathered} S \preceq M \implies \\ \forall \sigma_1 \xrightarrow{\tau_1} \sigma'_1 \xrightarrow{c_1} \sigma_2 \ldots \sigma_{n-1} \xrightarrow{\tau_{n-1}} \sigma'_{n-1} \xrightarrow{c_{n-1}} \sigma_n \in [\![S]\!] \centerdot \\ \forall i \in [1, n] \centerdot \sigma_i \models Inv(M) \wedge \forall i \in [1, n-1] \centerdot \tau_i \models Inv(M) \end{gathered}$$

However, checking $S \preceq M$ directly is complex because $S$ and $M$ combine discrete behavior by the application with continuous behavior by the controller. In the next section, we show how $S \preceq M$ can be checked compositionally using two separate verification steps – one for $A \preceq M$ and another for $C \preceq M$, and how each of these can be achieved using domain-specific verification tools.

## 3.4 Compositional Refinement Check

We now present our main theorem in the form of an assume-guarantee style proof rule.

Theorem 1 (Compositional Refinement).

$$\frac{A \preceq M \qquad C \preceq M}{A \parallel C \preceq M}$$

Proof. Let $S = A \parallel C$, $A \preceq M$, $C \preceq M$, and $\pi \in [\![S]\!]$ be any execution of $S$. Let:

$$\pi = \sigma_1 \xrightarrow{\tau_1} \sigma'_1 \xrightarrow{c_1} \sigma_2 \ldots \sigma_{n-1} \xrightarrow{\tau_{n-1}} \sigma'_{n-1} \xrightarrow{c_{n-1}} \sigma_n$$

The degenerate case of an execution where the application never executes is taken care of as part of the base case below.

For $i \in [1,n]$, let us write $\sigma_{A,i}$ to mean $\sigma_i \downharpoonright V_C$. By condition **SS4** in Definition 7 we know that:

$$\sigma_{A,1} \xrightarrow{c_1} \sigma_{A,2} \xrightarrow{c_2} \sigma_{A,3} \ldots \sigma_{A,n-1} \xrightarrow{c_{n-1}} \sigma_n \in [\![A]\!]$$

From condition **SS3** in Definition 7, we know that:

$$\sigma_1 \xrightarrow{\tau_1} \sigma'_1, \sigma_2 \xrightarrow{\tau_2} \sigma'_2, \ldots, \sigma_{n-1} \xrightarrow{\tau_{n-1}} \sigma'_{n-1} \in [\![C]\!]$$

For $i \in [1, n-1]$, define $\tilde{\sigma}_i = \sigma'_i \downharpoonright V_P$. From condition **SS2** in Definition 7, we know that $\forall i \in [1, n-1] \centerdot \tilde{\sigma}_i = \sigma_{i+1} \downharpoonright V_P$.

We will now show by induction that $\exists l_2, \ldots, l_n$ such that $(I, \sigma_1) \xrightarrow{\tau_1} (I, \sigma'_1) \xrightarrow{c_1} (l_2, \sigma_2) \ldots (l_{n-1}, \sigma_{n-1}) \xrightarrow{\tau_{n-1}} (l_{n-1}, \sigma'_{n-1}) \xrightarrow{c_{n-1}} (l_n, \sigma_n) \in [\![M]\!]$. Then, our result follows directly from Definition 10.

*Base Case:* From Definition 9, we know that $\tau_1 \models Inv(I)$. Hence $(I, \sigma_1) \xrightarrow{\tau_1} (I, \sigma'_1) \in \delta(M)_C$. Also note that $\sigma'_1 = \sigma_{A,1} \oplus \tilde{\sigma}_1$, $\sigma'_1 \models Inv(I)$, and $\sigma_2 = \sigma_{A,2} \oplus \tilde{\sigma}_1$. Hence from Definition 8, we have $\exists l_2 \centerdot (I, \sigma'_1) \xrightarrow{c_1} (l_2, \sigma_2) \in \delta(M)_A$. From Definition 4, we have $(I, \sigma_1) \xrightarrow{\tau_1} (I, \sigma'_1) \xrightarrow{c_1} (l_2, \sigma_2) \in [\![M]\!]$.

*Inductive Step:* Suppose $\exists l_2, \ldots, l_m$ such that $(I, \sigma_1) \xrightarrow{\tau_1} (I, \sigma'_1) \xrightarrow{c_1} (l_2, \sigma_2) \ldots (l_{m-1}, \sigma_{m-1}) \xrightarrow{\tau_{m-1}} (l_{m-1}, \sigma'_{m-1}) \xrightarrow{c_{m-1}} (l_m, \sigma_m) \in [\![M]\!]$. Using the inductive hypothesis, and Definition 9, we know that $\tau_m \models Inv(l_m)$. Hence $(l_m, \sigma_m) \xrightarrow{\tau_m} (l_m, \sigma'_m) \in \delta(M)_C$. Again note that $\sigma'_m = \sigma_{A,m} \oplus \tilde{\sigma}_m$ and $\sigma_{m+1} = \sigma_{A,m+1} \oplus \tilde{\sigma}_m$. Hence from the inductive hypothesis and Definition 8, we have $\exists l_{m+1} \centerdot (l_m, \sigma'_m) \xrightarrow{c_m} (l_{m+1}, \sigma_{m+1}) \in \delta(M)_A$. From Definition 4, this means $(I, \sigma_1) \xrightarrow{\tau_1} (I, \sigma'_1) \xrightarrow{c_1} (l_2, \sigma_2) \ldots (l_m, \sigma_m) \xrightarrow{\tau_m} (l_m, \sigma'_m) \xrightarrow{c_m} (l_{m+1}, \sigma_{m+1}) \in [\![M]\!]$. This completes the proof. □

Note that our proof rule is not complete. Consider a controller $C$ that fails to maintain the invariant $Inv(l_2)$ when location $l_2$ is reached via function call $c_1$. Suppose it is composed with an application $A$ that never calls $c_1$, i.e., the application prevents the controller from reaching the bad state. In this case, the conclusion of our rule $A \parallel C \preceq M$ holds, but the premise $C \preceq M$ does not.

# 4. VERIFYING PROOF-RULE PREMISES

In this section, we illustrate how to discharge the two premises of the proof rule given in Theorem 1, i.e, $A \preceq M$ and $C \preceq M$.

## 4.1 Checking Application Refinement

We assume that the application $A$ is a C-language program with calls to *Func*. To check $A \preceq M$, we construct stub-functions for each $f \in Func$ that check the conditions in Definition 8. We then verify $A$ along with the stub-definitions of *Func* using an off-the-shelf software model checker. Our stub-functions for *Func* are non-deterministic. This is necessary since the conditions in Definition 8 involve quantifiers.

More specifically, we assume a software model checker that supports three features: (i) non-deterministic value `*`; (ii) `assume` – a function that blocks all executions that invoke it with a FALSE argument, typically used to model the environment under which a specific part of a program is executed; and (iii) `assert` – a function that aborts all executions that invoke it with a FALSE argument, typically used to detect the violation of safety properties.

All these features are supported by most state-of-the-art software model checkers. For example, the bounded model checker CBMC [17] supports non-determinism via return values of undefined functions, `assume` via a call to the function `__CPROVER_assume`, as well as `assert`. Consider a contract automaton $M = (S, I, T, Inv, L)$. The body of the stub function for each $f \in func$ is generated as follows:

(a) Introduce a global variable *loc* to track the current state of $M$; *loc* is initialized to $I$.

(b) For each transition $(l, l') \in T$ with $L(l, l') = (f, req, grd, U, rv)$ generate code that: (i) is executed only if $loc = l$; (ii) assigns non-deterministic values to $V_P$; (iii) `assume`-s $Inv(l)$; (iv) `assert`-s condition *req*; (v) `assume`-s condition *grd*; (vi) executes assignments in $U$; (vii) `assert`-s $Inv(l')$; (viii) updates *loc* to $l'$; and (ix) `return`-s *rv*.

*Example.* Figure 3 shows the stub functions for `update_setpoint` and `has_arrived` from our example contract automaton in Figure 2. We omit statements that have no effect (e.g., `assert`-ing or `assume`-ing TRUE). Note that the `assert(0)` at the end of each function ensures that the function is never called when the contract automaton is in an inappropriate state. Also, since there are two transition from state *wait* labeled by *arrived*, they are both allowed non-deterministically.

The following theorem expresses the correctness of our

```
enum Loc {hover, wait};
Loc loc = hover;

void update_setpoint(double x, double y) {
  pos = *; //-- assign non-deterministic value
  if (loc == hover) {
    assume(INV_hover); assert(REQ_hover_wait);
    spnxt = (x,y); assert(INV_wait);
    loc = wait; return;
  }
  assert(0);
}

_Bool has_arrived() {
  if (loc == wait) {
    pos = *;
    //-- non-deterministic choice between
    //-- two outgoing transitions from wait
    if (*) {
      assume(INV_wait);
      assume(|pos - spnxt| > (0.1,0.1));
      assert(INV_wait); loc = wait; return 0;
    } else {
      assume(INV_wait);
      assume(|pos - spnxt| <= (0.1,0.1));
      spcur = spnxt; assert(INV_hover);
      loc = hover; return 1;
    }
  }
  assert(0);
}
```

**Figure 3: Stub definitions for our example contract automaton; `INV_x` denotes invariant $Inv(x)$; `REQ_a_b` denotes the *req* component of label $L(a, b)$. The * is a non-deterministic choice.**

```
void A1() {
 for (int n=1;;++n) {
  update_setpoint(n,0);
  while(!has_arrived());
 }
}
```

```
void A2() {
 for (int n=1;;++n) {
  update_setpoint(n,0);
  while(has_arrived());
 }
}
```

**Figure 4: Two example applications where initially `spcur = spnxt = (0,0)`. `A1()` refines our example contract automaton; `A2()` does not.**

procedure.

THEOREM 2 (APPLICATION REFINEMENT CHECK). *The C-language program $A$ together with the stub definitions of functions in Func constructed as above has no executions that violate an assertion if and only if $A \preceq M$.*

PROOF. (Sketch) Consider any $\sigma_1 \xrightarrow{c_1} \sigma_2 \xrightarrow{c_2} \sigma_3 \ldots \sigma_n \xrightarrow{c_n} \sigma_{n+1} \in [\![A]\!]$. It can be shown that there exists a sequence of locations $l_1, \ldots, l_{n+1}$ that satisfy condition **AR** of Definition 8 if and only if the C-language program $A$ together with the stub definitions of *Func* executes a sequence of function calls $c_1, \ldots, c_n$ such that for $i \in [1, n]$ the value of `loc` when $c_i$ is called is $l_i$, and the final value of `loc` is $l_{n+1}$. □

*Example.* Figure 4 shows two possible example applications (note the real quadcopter code we use is significantly more complex). `A1()` refines our example $M$ and the program obtained by combining it with the stub definitions in Figure 3 does not violate any assertions. `A2()` does not refine our example $M$ and the program obtained by combining it with the stub definitions in Figure 3 violates an assertion when it first calls `update_setpoint(5,0)`, then calls `has_arrived()` which returns FALSE, and then calls `update_setpoint(10,0)`.

The application code for each quadcopter was written in a domain-specific language, called DMPL [13], for programming distributed real-time systems, which includes a C-language code generator. This feature was used to generate the C-language source for the application $A$. The stub definitions for functions `update_setpoint_x()`, `has_arrived_x()`, `update_setpoint_y()` and `has_arrived_y()` were created manually from $M$ as shown in Figure 3. They were also written in DMPL, and then converted automatically to C source code. The combined application and stub functions, consisting of about 1700 LOC, were then verified using CBMC. Since CBMC is a bounded model checker, and our application does not terminate, CBMC cannot verify properties over (logically) unbounded program executions by itself. Therefore, we manually created loop-invariants and verified them to be inductive using CBMC, thus enabling us to prove unbounded properties. Essentially, to prove that `I` is an invariant of a loop with body `B`, we verify the following program with CBMC – `HAVOC(); __CPROVER_assume(I); B; assert(I);` – where `HAVOC()` assigns all relevant variables non-deterministic values. Note that the semantics of C is untimed and purely logical, and is therefore appropriate for modeling application-triggered transitions. Using a laptop with a quad-core 2.9 GHz CPU and 16 GB of RAM, the check took about 3.5 seconds. These invariants were also strong enough to imply all the assertions in the code. This proves $A \preceq M$.

## 4.2 Checking Controller Refinement

We assume that the physical system and low-level controller $C$ are modeled together as a hybrid automaton [29]. To check $C \preceq M$, we construct a hybrid automaton $H_M$ using $M$ such that the composed hybrid automaton $C \parallel H_M$ reaches a forbidden error state if $C \not\preceq M$. We then use an off-the-shelf hybrid system reachability analysis tool to verify that the forbidden states are not reachable in $C \parallel H_M$. In order to do this we need a hybrid automaton model checker which supports: (i) forbidden state checking; (ii) transitions with may-semantics; and (iii) automaton composition.

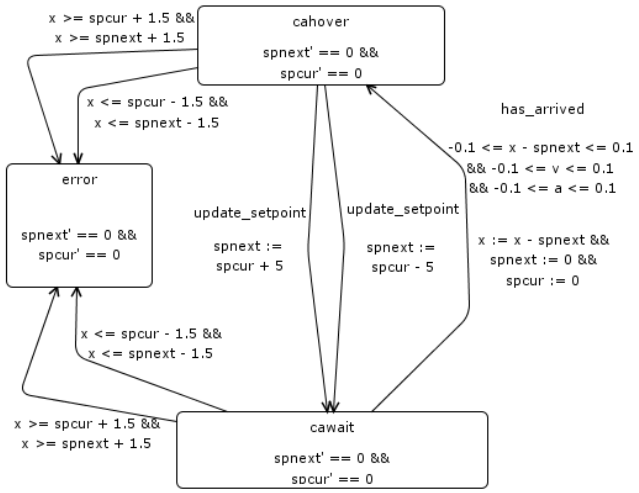



**Figure 5: Converted hybrid automaton extracted from contract automaton.**

These features are generally supported by hybrid systems model checkers like SPACEEX [28] or FLOW* [15]. Automaton composition can be performed by an external tool [4], if not supported natively by the reachability tool.

For 2-d position dynamics, we considered a simple, double-integrator system, where the position's derivative $\dot{x} = v$, the velocity's derivative $\dot{v} = a$, and the acceleration $a$ is the control input. We use a proportional-derivative (PD) low-level controller, with a proportional gain of 10 and a derivative gain of 3. Although hybrid systems reachability can handle more complicated dynamics and controllers, the focus of this work on the interface between the software and a hybrid automaton model. Thus, scalability and accuracy of hybrid systems reachability analysis becomes an orthogonal problem.

The $H_M$ derived from the contract automaton $M$ is given in Figure 5. The process of creating this automaton consists of first directly extracting the invariants and application guarantees from the original contact automaton. Next, the model is converted into a form amenable to analysis by a reachability tool, which consists of things like converting disjunctions in guards to multiple transitions, using compound conditions instead of min/max functions, and eliminating circular stutter transitions which do not affect analysis. These steps could be automated in a model transformation framework [4]. The last step involves reasoning about model symmetry in order to facilitate detection of fixpoints within reachability analysis. If the $x$ direction, for example, was unbounded, then the reachable set of states would be infinite, and reachability using flow-pipe construction would not complete. We take advantage of dynamics symmetry in the $x$ and $y$ directions in order to reduce the analysis to a single dimension, and furthermore, recenter the system to 0 whenever the controller settles near a new setpoint (the transition with the `has_arrived` label the $H_M$ automaton in Figure 5 has its reset assignment changed from `spcur := spnext` to `x := x - spnext && spnext := 0 && spcur := 0`). This symmetry reduction step needs to be proven correct, for example by using reachability reduction transformations [5].

The quadcopter low-level PD controller performs high-

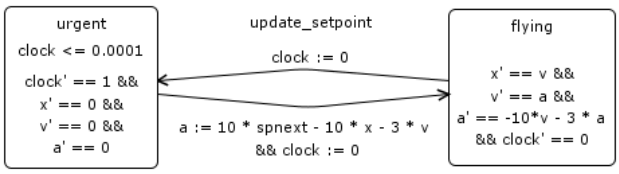
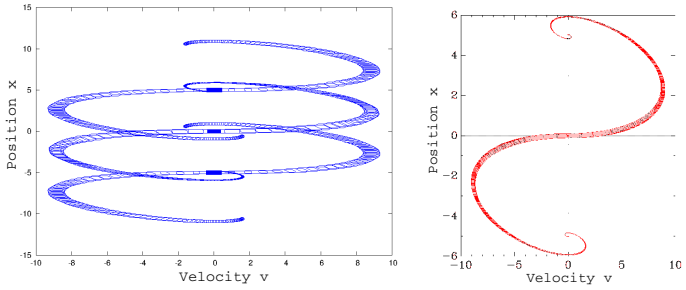



**Figure 6: The SpaceEx model of the continuous approximation of sampled quadcopter dynamics. The modes specify invariants and ODEs, while transitions have guards and instantaneous reset assignments.**




(a) (b)

**Figure 7: (a) Time-bounded reachability of the composed hybrid automaton, without symmetry reduction; (b) Reachability computation with symmetry reduction, which reaches a fixpoint.**

frequency sampling and actuation, subject to a real-time scheduler such as Rate Monotonic or Earliest-Deadline First [36], which guarantees one execution per period, but is not strictly periodic. Such a system can be abstracted for reachability analysis by a continuous one with an extra noise term in the dynamics, to account the effects of any possible jitter in the schedule [6]. We use the continuous approximation of the controller in our hybrid automaton model, shown in Figure 6.

A hybrid systems reachability tool can be used to analyze the hybrid automaton model consisting of the composition of the extracted automaton in Figure 5 and the controller/plant given in Figure 6. Without symmetry reduction, a time-bounded analysis can be performed. The projection of the reach set in terms of $x$ position and velocity, computed with Flow*, is shown in Figure 7(a). After changing the reset upon reaching the setpoint, as described above, a fixpoint in the reachable states can be detected. On a 2.3 GHz quad-core CPU laptop with 16 GB of memory, SpaceEx took about 33 seconds, 17 of which was the reachability computation. A 2-d projection of the reach set output by the tool is given in Figure 7(b). Neither of these models reach the `error` mode in the extracted contract automaton from Figure 5, which means that the contract's invariants are always met for the given plant/controller combination, and therefore $C \preceq M$. An example of a plant/controller which does not refine $M$ would be one with an unstable controller. Such as system would not have a response in the necessary bounds given in the invariants, and therefore reachability analysis would detect that the `error` mode is reachable.

# 5. PROVING HIGH-LEVEL PROPERTIES

The contract automaton method enables the proving of cyber-physical properties, which deal with individual quadcopters. The cyber-physical property $\Phi$ consists of the disjunction of the invariants of each of the modes of the contract automaton $M$, shown before in Figure 2. By Theorem 1, we have proven $\Phi$ holds for our quadcopter system, by checking $A \preceq M$ (done in Section 4.1) and $C \preceq M$ (done in Section 4.2). Now, we go beyond proofs of properties of individual quadcopters. We illustrate one way to use the CPS property $\Phi$ with additional formal verification techniques in order to perform end-to-end reasoning about collision avoidance between multiple quadcopters.

In particular, we want to show collision avoidance in a group of quadcopters in a finite, shared space. Specifically, we consider a system consisting of 10 quadcopters moving on a $100 \times 100$ 2-d area (i.e., $20 \times 20$ cells). As mentioned before, the quadcopter logic was programmed in DMPL. In addition to allowing C-language code generation, DMPL also allows use of the synchronous model of computation as a primitive in algorithm design. The code generated from DMPL uses a barrier-based protocol [11], built on top of the MADARA [25] middleware, to implement this synchronous model of computation. Also, DMPL's semantics takes care of packet loss and out-of-order arrival in the communication layer. The code generated from DMPL uses message retransmission and MADARA's packet reordering to remedy these situations.

Our system executes a synchronous distributed collision avoidance protocol. Each quadcopter maintains a cell variable *cellcur* corresponding to the current setpoint, and a cell variable *cellnext* corresponding to the destination setpoint. Each cell is treated as a shared resource, and a quadcopter always "locks" a cell by communicating with the others before moving into it. The synchronous model of computation is used to implement this distributed locking.

We refer to the 10 quadcopters as $N_0, N_1, \ldots, N_9$. Each quadcopter has its own copy of cyber and physical variables. For any such variable $x \in V_C \cup V_P$, we use $x[i]$ to denote the copy of $x$ for quadcopter $N_i$. Thus, for example, $spcur_x[2]$ is the $x$ coordinate of the current setpoint of $N_2$ and $pos_y[3]$ is the $y$ coordinate of the current position of $N_3$.

To prove collision avoidance, one property we need is that the cells defined by $cellcur[i]$ and $cellnext[j]$ are always mutually disjoint for distinct quadcopters, i.e.,

$$\forall 0 \leq i < j < 10 \centerdot \\ cellcur[i] \neq cellcur[j] \wedge cellcur[i] \neq cellnext[j] \wedge \\ cellnext[i] \neq cellcur[j] \wedge cellnext[i] \neq cellnext[j]$$

Note that this means essentially proving the correctness of the distributed locking algorithm. A second property to check is that for every quadcopter, the setpoints are 5 times the corresponding integer cell ids,

$$\forall 0 \leq i < 10 \centerdot 5 * cellcur[i] = (spcur_x[i], spcur_y[i]) \wedge \\ 5 * cellnext[i] = (spnxt_x[i], spnxt_y[i])$$

The verification step for these two properties leverages the synchronous model of computation provided by DMPL. The collision avoidance logic for all 10 quadcopters is combined into a single C-language program using the sequentialization technique [11], where computation proceeds in rounds based on the guarantees provided by the MADARA middleware. The combined program consists of about 17.5 KLOC,

| Potential Error | Detection |
|---|---|
| Software bug modifies setpoint twice in a row | SW |
| Software bug changes setpoint by both x and y | SW |
| Controller's gains are too high causing quadcopter to overshoot into neighboring cell | HY |
| Controller logic unstable | HY |
| Real-time period of low-level controller too low | HY |
| `has_arrived` condition too aggressive | HY |
| Barrier synchronization incorrectly used in communication protocol | DIST |
| Software does not reason about loss of communication | DIST |
| Buffer distances in cells too small | SMT |
| Helicopters too large for a given grid size | SMT |

**Table 1: A list of possible design and implementation errors, and where our approach would detect them. The detection locations are Software Model Checking (SW), Hybrid Systems Reachability (HY), Distributed System Sequentialization (DIST), and High-Level SMT Proof (SMT).**

about 10 times the size of the single-quadcopter application refinement check, which is then verified using CBMC. On same 2.9 GHz laptop that was used for the application refinement check, verification requires about 1900 seconds.

Given these three properties (the cyber-physical property from the contract automaton $\Phi$ and the two properties proven using sequentialization of the distributed system), we can now prove global collision avoidance. The three properties were formally written using SMT syntax, and as well as an additional assertion which encodes the condition under which a collision occurs (the positions are within twice the helicopter radius). This condition in SMT syntax is:

```
(<= (abs (- (pos i) (pos j))) (* 2.0 HELI_RADIUS))
```

Here, `i` and `j` are the x positions of two non-identical quadcopters (the check for y positions is similar). The satisfiability of these combined properties was then checked using Z3 [22], taking a fraction of a second. If $\texttt{HELI\_RADIUS} < 1.0$, the SMT solver returned `unsat`, indicating that no configuration is possible where all the properties are true and a collision is occurring. If $\texttt{HELI\_RADIUS} \geq 1.0$, then the SMT solver can produce counter-examples demonstrating a collision may be possible. For example, a possible counter-example has one quadcopter moving along the x direction from cell 0 to cell 1, and another quadcopter moving from cell 3 to cell 2. In this case, first quadcopter may be at position 6.5, while the second is at position 8.5 (recall they are permitted to deviate from the setpoints by up to 1.5 units). In this case, the quadcopters are exactly $2 * \texttt{HELI\_RADIUS}$ apart (when $\texttt{HELI\_RADIUS} = 1.0$).

To the best of our knowledge, this is the first formal verification of a distributed cyber-physical system that includes both the application software and the controller, using a sound combination of software model checking and hybrid reachability analysis. The proof includes end-to-end formal reasoning without gaps between analysis approaches, except for syntactic translations of properties, which could be automated (this translation would then ideally be proven correct). This makes it capable of catching a large variety of design and implementation mistakes. An outline of possible system errors, and where they would be detected using the proposed approach, is provided in Table 1.

# 6. RELATED WORK

*Software Model Checking.* Our work is complementary to, and leverages, verification techniques [31] for sequential C-language programs [7]. Sequentialization has been used for concurrent program verification. However, most of this work is targeted toward multi-threaded software [34, 20] or real-time software [12] executing on a single processor, not distributed applications. There has also been work on verifying distributed algorithms [32], while our goal is to verify distributed cyber-physical systems where each node has both discrete applications and hybrid components.

*Hybrid-systems verification* targets systems modeled using hybrid automata [1], which are best suited for modeling physical aspects of CPS with simpler discrete behaviors. Hybrid automata consist of, roughly, finite state machines combined with differential equations within each mode. Various hybrid automata model checkers exist depending on the complexity of the differential equations. Tools for computing reachability exist for timed automata [41], linear hybrid automata [28], and systems with general, nonlinear dynamics [15]. Other analysis methods for hybrid systems include falsification [23, 2], where the goal is to search for concrete inputs that lead to a property-violating trace.

*Composition Verification.* Assume-guarantee reasoning was proposed in the context of distributed programs [33], networks of processes [37], and program verification [39]. L* has been used [19, 14] to learn assumptions automatically. Compositional verification techniques have also been explored for model checking [18], probabilistic system verification [21], component-based reasoning with reals[16, 42], and hybrid systems [9, 8]. Within hybrid systems, analysis tractability can be improved by analyzing local components, and then reasoning separately about their composition [30, 3, 27]. However, these approaches assume systems with semantically uniform components (e.g., finite state automata), while we handle systems with discrete and dense hybrid components. Nuzzo et al. construct contracts across different domains, such as Linear Temporal Logic and Signal Temporal Logic [38], but for multi-layer controller synthesis.

*Cross-Domain Reasoning.* Some research explores reasoning across domains by abstracting the system from one domain into the other, where all the reasoning is performed. For example, continuous systems controlled by periodic software controllers are analyzed by converting the continuous dynamics to equivalent software code which advances physical variables according to the solutions of the differential equations, which may not always be available [24]. The system is then analyzed using off-the-shelf software verifiers, which may not scale over long time horizons. Alternatively, the continuous dynamics is abstracted by maneuver automata [26], which are finite state machines with timing information, describing both trim conditions and transitions between them. Such models can be used to synthesize distributed control strategies using SMT solvers [40]. Combined models with imperative semantics for programs and differential equations has also been proposed [10], but their formal analysis remains difficult. Symbolic execution of C software has been used to generate counter-examples for hybrid systems that explore all execution paths [43]. The StarL framework [35] contains primitives, specifications, and Java code, that can be composed and reasoned manually with the PVS theorem prover. However, the code itself is not proven to conform to the formal PVS specifications.

# 7. CONCLUSION

We presented a method to verify end-to-end safety properties of distributed CPSs. The crucial step was proving cyber-physical properties, which required reasoning over a combined software system and a hybrid automaton model of the low-level controller and plant. We used a contract automaton (CA) to formally describe the correct behavior of the application (in terms of legal sequence of API function calls and their pre-post-conditions and return values) and the controller (in terms of invariants maintained by its continuous dynamics). A sound assume-guarantee style proof rule was used to decompose the verification into two parts – one that verifies the application against the CA using software model checking, and another that verifies the controller against the CA using hybrid systems reachability analysis. The approach avoids the composition of discrete (application) and continuous (controller) behavior, ameliorating state-space explosion. It also permits the use of domain-specific (software and hybrid automata) specialized verification tools. The subsequent domain-specific analysis is simpler than the original combined CPS analysis. We used our approach to verify physical collision avoidance between a group of communicating quadcopters in a 2-d space. Our end-to-end proof is entirely performed using formal verification tools, except for syntactic translations of properties along the tool boundaries, which could be automated.

# 8. REFERENCES


[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138, 1995.

[2] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan. *S-taliro: A tool for temporal logic falsification for hybrid systems*. Springer, 2011.

[3] L. Aştefănoaei, S. Bensalem, and M. Bozga. A compositional approach to the verification of hybrid systems. In *Theory and Practice of Formal Methods*. Springer, 2016.

[4] S. Bak, S. Bogomolov, and T. T. Johnson. HyST: A source transformation and translation tool for hybrid automaton models. In *Proc. of HSCC*, 2015.

[5] S. Bak, Z. Huang, F. A. T. Abad, and M. Caccamo. Safety and progress for distributed cyber-physical systems with unreliable communication. *ACM Trans. Embed. Comp. Sys.*, 14(4), 2015.

[6] S. Bak and T. T. Johnson. Periodically-scheduled controller analysis using hybrid systems reachability and continuization. In *Proc. of RTSS*, 2015.

[7] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proc. of SPIN*, 2001.

[8] L. Benvenuti, A. Ferrari, L. Mangeruca, E. Mazzi, R. Passerone, and C. Sofronis. A contract-based formalism for the specification of heterogeneous systems. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 142–147, Sept 2008.

[9] S. Bogomolov, G. Frehse, M. Greitschus, R. Grosu, C. S. Pasareanu, A. Podelski, and T. Strump. Assume-Guarantee Abstraction Refinement Meets Hybrid Systems. In *Proc. of HVC*, 2014.

[10] O. Bouissou. From control-command synchronous programs to hybrid automata. In *Proc. of ADHS*, 2012.

[11] S. Chaki and J. Edmondson. Model-Driven Verifying Compilation of Synchronous Distributed Applications. In *Proc. of MODELS*, 2014.

[12] S. Chaki, A. Gurfinkel, and O. Strichman. Time-Bounded Analysis of Real-Time Systems. In *Proc. of FMCAD*, 2011.

[13] S. Chaki and D. Kyle. DMPL: Programming and verifying distributed mixed-synchrony and mixed-critical software. Technical Report CMU/SEI-2016-TR-005, 2016. resources.sei.cmu.edu/library/asset-view.cfm?assetid=464254.

[14] S. Chaki and N. Sinha. Assume-Guarantee Reasoning for Deadlock. In *Proc. of FMCAD*, 2006.

[15] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *Proc. of CAV*, 2013.

[16] A. Cimatti and S. Tonetta. Contracts-refinement proof system for component-based embedded systems. *Sci. Comput. Program.*, 97:333–348, 2015.

[17] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, 2004.

[18] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proc. of LICS*, 1989.

[19] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning Assumptions for Compositional Verification. In *Proc. of TACAS*, 2003.

[20] L. Cordeiro and B. Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *Proc. of ICSE*. Association for Computing Machinery, 2011.

[21] L. de Alfaro, T. A. Henzinger, and R. Jhala. Compositional Methods for Probabilistic Systems. In *Proc. of CONCUR*, 2001.

[22] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, 2008.

[23] A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Proc. of CAV*, 2010.

[24] P. S. Duggirala and M. Viswanathan. Analyzing real time linear control systems using software verification. In *Proc. of RTSS*, 2015.

[25] J. R. Edmondson and A. S. Gokhale. Design of a Scalable Reasoning Engine for Distributed, Real-Time and Embedded Systems. In *Proc. of KSEM*, 2011.

[26] E. Frazzoli, M. A. Dahleh, and E. Feron. Maneuver-based motion planning for nonlinear systems with symmetries. *Robotics, IEEE Transactions on*, 21(6), 2005.

[27] G. Frehse. Compositional verification of hybrid systems with discrete interaction using simulation relations. In *Proc. of CACSD*, 2004.

[28] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable Verification of Hybrid Systems. In *Proc. of CAV*, 2011.

[29] T. A. Henzinger. The Theory of Hybrid Automata. In *Proc. of LICS*, 1996.

[30] Z. Huang and S. Mitra. Proofs from simulations and modular annotations. In *Proc. of HSCC*, 2014.

[31] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4), 2009.

[32] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Proc. of FMCAD*, 2013.

[33] C. B. Jones. Specification and Design of (Parallel) Programs. In *Proc. of 9th IFIP World Congress*, 1983.

[34] A. Lal and T. W. Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *Proc. of CAV*, 2008.

[35] Y. Lin and S. Mitra. Starl: Towards a unified framework for programming, simulating and verifying distributed robotic systems. In *Proc. of LCTES*, 2015.

[36] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM*, 20(1), 1973.

[37] J. Misra and K. M. Chandy. Proofs of Networks of Processes. *TSE*, 7(4), 1981.

[38] P. Nuzzo, H. Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. DonzÃl', and S. A. Seshia. A contract-based methodology for aircraft electric power system design. *IEEE Access*, 2:1–25, 2014.

[39] A. Pnueli. In Transition from Global to Modular Temporal Reasoning About Programs. *Logics and Models of Concurrent Systems*, 13, 1985.

[40] I. Saha, R. Ramaithitima, V. Kumar, G. J. Pappas, and S. A. Seshia. Automated composition of motion primitives for multi-robot systems from safe ltl specifications. In *Proc. of IROS*, 2014.

[41] A. University and U. University. Uppaal - a tool suite for verification of real-time systems. http://www.uppaal.com, 2008.

[42] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. P. E. Heimdahl, and S. Rayadurgam. Your what is my how: Iteration and hierarchy in system design. *IEEE Software*, 30(2):54–60, March 2013.

[43] A. Zutshi, S. Sankaranarayanan, J. Deshmukh, and X. Jin. Symbolic-numeric reachability analysis of closed-loop control software. In *Proc. of HSCC*, 2016.