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;

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.,
Theorem 3.1. Let $A$ and $C$ be properties of a system, and let $M$ be a set of observations. Then $A \vdash C$ if and only if $A \supset C$.

Proof. ($\Rightarrow$) Suppose $A \vdash C$. Then there exists a proof sequence $A_0, A_1, \ldots, A_n = C$ such that $A_{i-1} \vdash A_i$ for all $i$. We can construct a proof of $A \supset C$ by setting $A_i$ as the property $A_{i-1}$ at each step.

($\Leftarrow$) Suppose $A \supset C$. Then for all $A$, if $A \vdash C$, then $A$ must be true. This is precisely the definition of $A \vdash C$.

We can use this proof theorem to prove the correctness of a system. For example, we can prove that the system will not crash if the input is valid.
The application always calls `update_setpoint(x, y)`, with arguments that satisfy the condition \(|(x, y) - spcur| = (5, 0) \lor |(x, y) - spcur| = (0, 5)\).

(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., \(|spnxt - spcur| = (5, 0) \lor |spnxt - spcur| = (0, 5)\)), the controller must maintain the following invariant:

\[ \Phi_{move} \equiv \min(spcur_x, spnxt_x) - 1.5 \leq pos_x \]
\[ \leq \max(spcur_x, spnxt_x) + 1.5 \]
\[ \land \min(spcur_y, spnxt_y) - 1.5 \leq pos_y \]
\[ \leq \max(spcur_y, spnxt_y) + 1.5 \]

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 \( V \) to denote the set of all variables, i.e., \( V = V_c \cup V_{par} \cup V_p \). 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` 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 \( V \subseteq V \), we write \( \text{Expr}(V) \) to denote a set of expressions constructed from \( V \cup \mathbb{D} \), using numeric operators (+, −, *, /, etc.), relational operators (\( <, \leq, >, \geq \), etc.), and logical operators (\( \land, \lor, \neg \), 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 \( \text{Func} \). 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, \( a : p \rightarrow \mathbb{D} \) 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 \( \text{Func} = \{ f_1, f_2 \} \) where: (i) \( f_1 = (\text{update_setpoint}, (x, y), \text{spcur}) \); and (ii) \( f_2 = (\text{has_arrived}, (), \text{spcur}) \). Some possible function calls are \((f_1, (1, 1), \text{spcur})\), \((f_2, (), \text{spcur}, \text{spnxt})\), etc.

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

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 \((\text{lhs}, \text{rhs})\) where \( \text{lhs} \in V \) is the left-hand side and \( \text{rhs} \in \text{Expr}(V) \) is the right-hand side. The set of all assignments is \( \text{Asgn} \).

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

- \( S \) is a finite set of locations;
- \( I \subseteq S \) is the initial location;
- \( T \subseteq S \times S \) is a transition relation;
- \( \text{Inv} : S \rightarrow \text{Expr}(V_p \cup V_c) \) maps each location to an expression over the physical and cyber variables; informally, \( \text{Inv}(l) \) is the invariant that a correct controller should maintain when the system is in location \( l \);
- \( L : T \rightarrow \text{Func} \times \text{Expr} \times \text{Expr} \times \text{Asgn}^* \times (\mathbb{D} \cup \{ \} \) 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 \( L(l, l') = (f, \text{req}, \text{grd}, U, rv) \), then it means:

- The transition from \( l \) to \( l' \) is triggered by a call to function \( f = (f_n, p, rt) \) by the application.
- Any such call must satisfy the condition \( \text{req} \), which is an expression over \( V_f \cup p \).
- Once the transition is triggered, it can only occur if condition \( \text{grd} \), which is an expression over \( V_f \cup V_c \cup p \), holds. The key difference between \( \text{req} \) and \( \text{grd} \) is that while every call to \( f \) by \( A \) must satisfy \( \text{req} \), it does not have to satisfy \( \text{grd} \).
- 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 \( (\text{grd}) \) then \( \{ U; \text{return } rv\} \)

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 \( \sigma : V \to \mathbb{B} \) is a partial assignment of variables to values. The domain of \( \sigma \) is denoted \( \text{Dom}(\sigma) \). We write \( \sigma_1 \odot \sigma_2 \) to denote the state obtained by merging \( \sigma_1 \) and \( \sigma_2 \) with disjoint domains, i.e., if \( \text{Dom}(\sigma_1) \cap \text{Dom}(\sigma_2) = \emptyset \), then:

\[
(\sigma_1 \odot \sigma_2)(v) = \sigma_i(v), \text{ } v \in \text{Dom}(\sigma_i), i \in \{1, 2\}
\]

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

\[
\Sigma(V) = \{ \sigma : V \to \mathbb{B} \mid \text{Dom}(\sigma) = V \}
\]

Given a state \( \sigma \) and a set of variables \( V \subseteq \text{Dom}(\sigma) \), the projection of \( \sigma \) on \( V \), denoted \( \sigma|_V \), is the state such that:

\[
\text{Dom}(\sigma|_V) = V \land \forall v \in V, \sigma(v) = \sigma(v)
\]

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, \( [v_1 = 5, v_2 = 3, v_3 = 2] = 2 \), and \([v_1 = 2, v_2 = 3, v_3 = 2] = \text{false}\). 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 \( (\sigma|_V = (\sigma'|_V) \), 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, \((\forall \sigma'' \in \tau, \sigma'' \mid V_c) = (\sigma \mid V_c)\). Given expression \( e \), we write \( \tau \models e \) to mean \( \forall \sigma \in \tau, \sigma \models e \).

**Contract Automaton Transition.** Let \( M = (S, I, T, \text{Inv}, L) \) 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 \( \sigma \in \Sigma(V_f \cup V_c) \). There are two types of transitions – activation-triggered and controller triggered. An application-triggered transition is of the form \((l, \sigma) \xrightarrow{f} (l', \sigma')\) such that: (i) \( (l, l') \in T \); (ii) \( \sigma|_V = (\sigma'|_V) \); note that this means an application-triggered transition does not alter the values of physical variables; (iii) \( \sigma \models \text{Inv}(l) \land \sigma' \models \text{Inv}(l') \); and (iv) \( L(l, l') = (f, \text{req}, \text{grd}, U, rv) \) and \( e = (f, a, rv) \) such that:

- \( \sigma + a \models \text{req} \land \text{grd} \); 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 \((l, \sigma) \xrightarrow{c} (l', \sigma')\) such that: (i) \( \tau \) is a trajectory from \( \sigma \) to \( \sigma' \); (ii) \( \sigma|_V = (\sigma'|_V) \); and (iii) \( \sigma \models \text{Inv}(l) \); 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:

\[
(l_1, \sigma_1) \xrightarrow{t_1} (l_1, \sigma_1') \xrightarrow{c_1} (l_2, \sigma_2) \ldots (l_{n-1}, \sigma_{n-1}') \xrightarrow{c_{n-1}} (l_n, \sigma_n)
\]

such that:

- \( l_1 = I \land \forall i \in [1, n - 1], (l_i, \sigma_i) \xrightarrow{t_i} (l_i, \sigma_i') \in \delta(M)_C \land \forall i \in [1, n - 1], (l_i, \sigma_i') \xrightarrow{c_{i-1}} (l_{i+1}, \sigma_{i+1}) \in \delta(M)_A \)

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 \( \text{Func} \). 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., \( \forall i \geq 1, \text{Dom}(\sigma_i) = V_c \).

**Definition 6 (Controller Semantics).** Since the controller \( C \) is defined by a hybrid automaton, its semantics is given by a set of executions over \( V_c \cup V_f \), and an initial
Definition 7 (System Semantics). The system $S = A \parallel C$ is an asynchronous interleaving of the application and the controller where function calls are made by the application in response to events generated by the controller. The semantics of $S$, denoted $[S]$, is given by a set of execution sequences:

$$\sigma_1 \stackrel{c_1}{\rightarrow} \sigma_1' \stackrel{c_2}{\rightarrow} \sigma_2 \cdots \stackrel{c_{n-1}}{\rightarrow} \sigma'_{n-1} \stackrel{c_n}{\rightarrow} \sigma_n$$

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

(SS1) $\forall i \geq 1. \text{Dom}(\sigma_i) = \text{Dom}(\sigma'_i) = V_C \cup V_P$

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

(SS3) $\sigma_1 \stackrel{c_1}{\rightarrow} \sigma_1' \stackrel{c_2}{\rightarrow} \sigma_2 \cdots \stackrel{c_{n-1}}{\rightarrow} \sigma'_{n-1}\in [C]$

(SS4) $\sigma_i \mid V_C \stackrel{c_i}{\rightarrow} \sigma'_i \mid V_C \wedge \cdots \wedge \sigma_{n-1} \mid V_C \in [A]$

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:

$$A \preceq M \iff \forall \sigma \in [A]. \quad \forall \sigma_1 \rightarrow \sigma_1' \rightarrow \sigma_2 \cdots \rightarrow \sigma_{n-1} \rightarrow \sigma_n \rightarrow \in [C]. \quad \forall \sigma_i \mid V_C \rightarrow \sigma'_i \mid V_C \wedge \cdots \wedge \sigma_{n-1} \mid V_C \in [A].$$

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:

$$C \preceq M \iff \forall \sigma \in [A]. \quad \forall \sigma_1 \rightarrow \sigma_1' \rightarrow \sigma_2 \cdots \rightarrow \sigma_{n-1} \rightarrow \sigma_n \rightarrow \in [C]. \quad \forall \sigma_i \mid V_C \rightarrow \sigma'_i \mid V_C \wedge \cdots \wedge \sigma_{n-1} \mid V_C \in [A].$$

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:

$$S \preceq M \iff \forall \sigma_1 \rightarrow \sigma_1' \rightarrow \sigma_2 \cdots \rightarrow \sigma_{n-1} \rightarrow \sigma_n \rightarrow \in [S]. \quad \exists l_2, l_3, \ldots, l_n \cdot (l_2, l_3, \ldots, l_n) \rightarrow (l_2', l_3', \ldots, l_n') \in [M].$$

3.3 Cyber-Physical Properties

A contract automation’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 automation). The CPS properties are expressed as relations over the cyber and physical variables. For the contract automation from Figure 2, the corresponding cyber-physical property $\Phi$ is:

$$(\Phi_{move} \wedge spnxt = spcur) \vee (\Phi_{move} \wedge (spnxt \neq spcur = (5,0))$$

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

Proposition 1 (Invariant Satisfaction).

$$\forall (l_1, l_1') \rightarrow (l_2, l_2') \rightarrow \ldots \rightarrow (l_{n-1}, l_{n-1'}) \rightarrow (l_n, l_n') \in [M].$$

Proposition 2 (Invariant Preservation).

$$S \preceq M \iff \forall \sigma_1 \rightarrow \sigma_1' \rightarrow \sigma_2 \cdots \rightarrow \sigma_{n-1} \rightarrow \sigma_n \rightarrow \in [S]. \quad \forall \sigma_i \mid V_C \rightarrow \sigma'_i \mid V_C \wedge \cdots \wedge \sigma_{n-1} \mid V_C \in [A].$$

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).

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

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

$$\pi = \sigma_1 \rightarrow \sigma_1' \rightarrow \sigma_2 \cdots \rightarrow \sigma_{n-1} \rightarrow \sigma_n$$

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 $\sigma_i | V_C$. By condition SS4 in Definition 7 we know that:

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

From condition SS3 in Definition 7 we know that:

$$\sigma_1 \xrightarrow{r_1} \sigma_1', \sigma_2 \xrightarrow{r_2} \sigma_2', \ldots, \sigma_{n-1} \xrightarrow{r_{n-1}} \sigma_{n-1}' \in [C]$$

For $i \in [1, n-1]$, define $\sigma_i = \sigma_i' | V_p$. From condition SS2 in Definition 7 we know that $\forall i \in [1, n-1], \sigma_i = \sigma_{i+1} | V_p$.

We will now show by induction that $\exists l_2, \ldots, l_m$ such that $(I, \sigma_1) \xrightarrow{r_1} (I, \sigma_1') \xrightarrow{c_1} (l_2, \sigma_2) \cdots \xrightarrow{r_{n-1}} (l_{n-1}, \sigma_{n-1}') \xrightarrow{c_{n-1}} (l_n, \sigma_n) \in [M]$. Then, our result follows directly from Definition 10.

**Base Case:** From Definition 9, we know that $\tau_i \models Inv(I)$. Hence $(I, \sigma_1) \xrightarrow{r_1} (I, \sigma_1') \in \delta(M)_C$. Also note that $\sigma_1 = \sigma_1 | \hat{\sigma}_1$, $\sigma_1' = \sigma_1 | Inv(I)$, and $\sigma_2 = \sigma_{A,2} \oplus \hat{\sigma}_1$. Hence from Definition 8, we have $\exists l_2, (I, \sigma_1') \xrightarrow{c_2} (l_2, \sigma_2) \in \delta(M)_A$. From Definition 4, we have $(I, \sigma_1') \xrightarrow{c_2} (l_2, \sigma_2) \in [M]$.

**Inductive Step:** Suppose $\exists l_2, \ldots, l_m$ such that $(I, \sigma_1) \xrightarrow{r_1} (I, \sigma_1) \xrightarrow{c_1} (l_2, \sigma_2) \cdots \xrightarrow{r_{n-1}} (l_{n-1}, \sigma_{n-1}) \xrightarrow{c_{n-1}} (l_n, \sigma_n) \in [M]$. Using the inductive hypothesis, and Definition 9, we know that $\tau_m \models Inv(l_n)$. Hence $(l_n, \sigma_n) \xrightarrow{r_{n-1}} (l_{n-1}, \sigma_{n-1}) \in \delta(M)_C$. Again note that $\sigma_{n-1} = \sigma_{A,n-1} \oplus \hat{\sigma}_m$ and $\sigma_{n-1} = \sigma_{A,n-1} \oplus \hat{\sigma}_m$. Hence from the inductive hypothesis and Definition 8, we have $\exists l_{n-1}, (l_n, \sigma_n) \xrightarrow{c_{n-1}} (l_{n-1}, \sigma_{n-1}) \in \delta(M)_A$. From Definition 4, this means $(I, \sigma_1) \xrightarrow{r_1} (I, \sigma_1) \xrightarrow{c_1} (l_2, \sigma_2) \cdots \xrightarrow{r_{n-1}} (l_{n-1}, \sigma_{n-1}) \xrightarrow{c_{n-1}} (l_n, \sigma_n) \in [M]$. 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 $V_f$; (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**-$s$ or **assume**-$s$ TRUE). Note that the **assert**-$s$(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 transitions from state $wait$ labeled by arrived, they are both allowed non-deterministically.

The following theorem expresses the correctness of our
procedure.

**Theorem 2 (Application Refinement Check).**
The C-language program $A$ together with the stub definitions of functions in $\text{Func}$ constructed as above has no executions that violate an assertion if and only if $A \preceq M$.

**Proof. (Sketch)** Consider any $\sigma_1 \xrightarrow{c_1} \sigma_2 \xrightarrow{c_2} \ldots \xrightarrow{c_n} \sigma_{n+1} \in [A]$. 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 $\text{Func}$ executes a sequence of function calls $c_1, \ldots, c_n$ such that for $i \in [1, n]$ the value of $\text{loc}$ when $c_i$ is called is $l_i$, and the final value of $\text{loc}$ 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 $\text{update_setpoint}(5,0)$, then calls $\text{has_arrived}()$ which returns $\text{false}$, and then calls $\text{update_setpoint}(10,0)$.

The application code for each quadcopter was written in a domain-specific language, called $\text{DMPL}$ [13], for programming distributed real-time systems, which includes a C-language code generator. This feature was used to generate the C-language source code for the application $A$. The stub definitions for functions $\text{update_setpoint}_x()$, $\text{has_arrived}_x()$, $\text{update_setpoint}_y()$ and $\text{has_arrived}_y()$ were created manually from $M$ as shown in Figure 3. They were also written in $\text{DMPL}$, 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 – $\text{HAVOC}()$; $\text{CPROVER}_\text{assume}(I)$; $B$; $\text{assert}(I)$; - where $\text{HAVOC}()$ assigns all relevant variables non-deterministic values. Note that the semantics of $I$ 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. These features are generally supported by hybrid systems model checkers like SPACEX [28] or FLOW* [15].

Figure 5: Converted hybrid automaton extracted from contract automaton.

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 contract 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 fixedpoints 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 $\text{has_arrived}$ label the $H_M$ automaton in Figure 5 has its reset assignment changed from $\text{spcur} := \text{spnext}$ to $x := x - \text{spnext}$ & $\text{spnext} := 0$ & $\text{spcur} := 0$). This symmetry reduction step needs to be proven correct, for example by using reachability reduction transformations [5].
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 \( A \leq M \) (done in Section 4.1) and \( C \leq M \) (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 \( \text{cellcur} \) corresponding to the current setpoint, and a cell variable \( \text{cellnext} \) 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, \( \text{spcur}_{i,2} \) is the \( x \) coordinate of the current setpoint of \( N_2 \) and \( \text{pos}_{i,3} \) 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 \( \text{cellcur}[i] \) and \( \text{cellnext}[j] \) are always mutually disjoint for distinct quadcopters, i.e.,

\[
\forall 0 \leq i < j < 10, \\
\text{cellcur}[i] \neq \text{cellcur}[j] \land \text{cellcur}[i] \neq \text{cellnext}[j] \\
\text{cellnext}[i] \neq \text{cellcur}[j] \land \text{cellnext}[i] \neq \text{cellnext}[j]
\]

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,

\[
\forall 0 \leq i < 10, 5 \times \text{cellcur}[i] = (\text{spcur}_{i,3}, \text{spcur}_{i,2}) \\
5 \times \text{cellnext}[i] = (\text{spnext}_{i,3}, \text{spnext}_{i,2})
\]

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.

---

### 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.

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

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

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 Φ 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:

$$ (\leq (\text{abs} (- (\text{pos} i) (\text{pos} j))) (* 2.0 \text{ HELI}\_\text{RADIUS})) $$

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 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 HELI\_RADIUS ≥ 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 * HELI\_RADIUS apart (when 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