2008 American Control Conference
Westin Seattle Hotel, Seattle, Washington, USA
June 11-13, 2008

Model Checking In-The-Loop: Finding Counterexamples by Systematic Simulation

Flavio Lerda[†], James Kapinski[§], Hitashyam Maka[§], Edmund M. Clarke[†], and Bruce H. Krogh[§]
[†] School of Computer Science
[§] Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
Email: [†]{flerda|emc}@cs.cmu.edu, [§]{jpk3|hmaka|krogh}@andrew.cmu.edu

Abstract— **Model checkers for program verification have enjoyed considerable success in recent years. In the control systems domain, however, they suffer from an inability to account for the physical environment. For control systems, simulation is the most widely used approach for validating system designs. We present a new technique for finding counterexamples that uses a software model checker to perform a systematic simulation of the software implementation of a controller coupled with a continuous plant. Instead of performing a large set of independent simulations, our approach uses the model checking notion of state-space exploration by piecing together numerical simulations of the plant and transitions of the controller. Our implementation of this technique uses an explicit-state source-code model checker to analyze the software and the MATLAB/Simulink environment to model and simulate the plant. We present an illustrative example involving a supervisory controller for an unmanned aerial vehicle (UAV). We show that our technique is able to detect an error in the controller design.**

I. Introduction

The goal of model-based design of embedded software is to reduce development time and cost by evaluating controllers using computer-based models. This approach requires methods for exploring the behaviors of dynamical systems. While simulation can be used to evaluate system performance for a specific set of parameters, exhaustive evaluation of system behaviors over a range of parameters using simulation is usually intractable. We investigate using formal verification techniques to catch design errors by exploring the behaviors of an embedded control system. Our approach combines a software model checker with numerical simulation.

Model checking is an automated technique for checking properties of finite-state systems [7], [14], [8]. If a given property to be verified is not true, the model checking algorithm produces a *counterexample*, which is a trace of state transitions in the finite state system that violates the property. In recent years, there has been considerable interest in model checkers for software [3], [10], [5], [15]. One of the main advantages of model checking compared to other validation techniques, such as simulation, is the ability to explore the behaviors of a system exhaustively.

Embedded control systems are difficult to analyze using model checking due to the controller's interaction with a continuous dynamic plant, which makes the system infinite state. Various methods have been developed to formally verify hybrid automata, which can be used to model embedded control systems [9], [12], [4], [6]. These techniques are computationally expensive, however, and are able to analyze only systems of low complexity.

Simulation is the most widely used technique for validating control system designs. Tools like MATLAB/Simulink provide an environment for modeling and simulating control systems [13], [2]. Recently, a MATLAB toolbox for verification was released, the *Design Verification Toolbox* [1], but this tool only addresses the verification of discrete-time components of Simulink models.

We present an approach that narrows the gap between simulation and model checking of control systems. Using a numerical simulator, we can capture the dynamics of the plant accurately. By employing a model checker, we can validate properties that are difficult to validate using standard simulation, such as correctness of an implementation using concurrent tasks communicating via shared variables. The technique is implemented using an explicit-state source-code model checker and the MATLAB/Simulink simulation environment. We present an example based on the Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC) platform [11].

II. System Model

We consider a sampled-data controller and a continuous-time plant. A *sampled-data controller* is able to observe the state of the plant only at discrete time instants, called *sample times*. We assume that the sample times occur at multiples of a fixed *sampling period*, t_s. The software that implements the controller is composed of a set of concurrent tasks. The tasks execute periodically at the sample times. We assume that the code of the controller executes instantaneously and all tasks share the same clock. The plant is modeled as a set of differential equations. Figure 1 shows the architecture of a sampled-data system. For simplicity, we show the controller

This research was sponsored by the Air Force Research Office (AFRO) under contract no. FA9550-06-1-0312, by the General Motors Collaborative Research Lab at Carnegie Mellon University under grant no. GM9100096UMA, and by the National Science Foundation (NSF) under grant no. CCR-0411152.

(resp. plant) observing the entire state of the plant (resp. controller), however, our analysis does not require this. The controller can observe a mapping $g(\mathbf{x})$ of the plant state and the plant can observe a mapping $h(\mathbf{v})$ of the controller state.

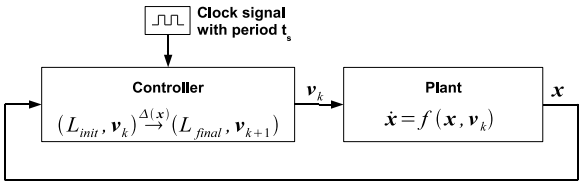


Fig. 1. The architecture of a sampled-data system.

We use a single finite-state automaton to describe the behavior of the controller, which represents a set of concurrently executing tasks. In the following, we describe how the finite-state automaton for the controller can be obtained from the set of finite-state automata representing the tasks.

Let $\mathbf{v} \in V^m$ be the controller variables shared among the tasks, where V is a finite set of values. Each controller task T_i is described as a finite-state automaton with shared variables. Let Loc_i be the states of the automaton, called *local control locations*. Let $l_{i,\text{init}}, l_{i,\text{final}} \in Loc_i$ be two specially designated control locations. At each sample time, task T_i starts at control location $l_{i,\text{init}}$ and ends at control location $l_{i,\text{final}}$. Let $Q_i = Loc_i \times V^m$ represent the local controller states of task T_i. The transitions of the task depend on the state of the plant, $\mathbf{x} \in \mathbb{R}^n$. The local transition relation of T_i is defined by $\delta_i : \mathbb{R}^n \rightarrow 2^{Q_i \times Q_i}$. There exists a local controller transition of T_i denoted by $(l_i, \mathbf{v}) \xrightarrow{\delta_i(\mathbf{x})} (l_i', \mathbf{v}')$ if and only if $((l_i, \mathbf{v}), (l_i', \mathbf{v}')) \in \delta_i(\mathbf{x})$. No transition is possible from the final control location $l_{i,\text{final}}$.

Given a set of tasks $T_1, \ldots, T_p$, the controller is obtained by composing the tasks using *interleaving semantics*, meaning that there is no predetermined execution order between the transitions of different tasks. Interleaving semantics is used to model control software that is either implemented as multiple threads or distributed among a set of processors.

The composed system is represented as a finite-state automaton. Let $Loc = Loc_1 \times \ldots \times Loc_p$ be the states of the automaton, called the *control locations*. Let $L_{\text{init}} = (l_{1,\text{init}}, \ldots, l_{p,\text{init}})$ and $L_{\text{final}} = (l_{1,\text{final}}, \ldots, l_{p,\text{final}})$ be the initial and final control locations. Let $Q = Loc \times V^m$ represent the controller states. The controller transition relation is defined by $\delta : \mathbb{R}^n \rightarrow 2^{Q \times Q}$ such that $(((l_1, \ldots, l_p), \mathbf{v}), ((l_1', \ldots, l_p'), \mathbf{v}')) \in \delta(\mathbf{x})$ if and only if there exists a task T_i such that $(l_i, \mathbf{v}) \xrightarrow{\delta_i(\mathbf{x})} (l_i', \mathbf{v}')$ and $\forall j \neq i : l_j = l_j'$. Given two controller states $q, q' \in Q$ and a plant state $\mathbf{x}$, we denote $(q, q') \in \delta(\mathbf{x})$ by $q \xrightarrow{\delta(\mathbf{x})} q'$.

At each sample time, the controller starts executing at control location L_{init} and stops when it reaches control location L_{final}. We can define a transition relation $\Delta : \mathbb{R}^n \rightarrow 2^{Q \times Q}$ that represents the relation between the states of the controller at the beginning and at the end of each execution. Given a plant state $\mathbf{x} \in \mathbb{R}^n$, we have that $((L_{\text{init}}, \mathbf{v}), (L_{\text{final}}, \mathbf{v}')) \in \Delta(\mathbf{x})$, also denoted as $(L_{\text{init}}, \mathbf{v}) \xrightarrow{\Delta(\mathbf{x})} (L_{\text{final}}, \mathbf{v}'))$, if and only if there exists a finite sequence of controller states $q_0 \ldots q_J$ such that $q_0 = ((L_{\text{init}}, \mathbf{v})$, $q_J = (L_{\text{final}}, \mathbf{v}')$, and $q_j \xrightarrow{\delta(\mathbf{x})} q_{j+1}$ for every $0 \leq j < J$.

A *sampled-data control system* is a tuple $SDCS = (Loc, L_{\text{init}}, L_{\text{final}}, V, \delta, f, t_s, Init)$, where:

- Loc is a finite set of control locations;
- $L_{\text{init}}, L_{\text{final}} \in Loc$ are the first and last control locations of each periodic execution of the controller;
- V is a finite set of values;
- $\delta : \mathbb{R}^n \rightarrow 2^{Q \times Q}$ is the controller transition relation;
- $f : \mathbb{R}^n \times V^m \rightarrow \mathbb{R}^n$ is Lipschitz continuous in its first argument and describes the continuous flow of the system as a function of the controller variables;
- $t_s > 0$ is the sampling period of the controller. The controller executes only at time instances corresponding to non-negative multiples of t_s;
- $Init \subset Loc \times V^m \times \mathbb{R}^n$ is a finite set of initial states.

Let $S = Loc \times V^m \times \mathbb{R}^n$ denote the set of system states. Evolutions of the plant over a sampling period t_s are defined implicitly by the set of differential equations $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{v})$. Let $F : V^m \rightarrow 2^{\mathbb{R}^n \times \mathbb{R}^n}$ be the discrete-time update function, such that $(\mathbf{x}, \mathbf{x}') \in F(\mathbf{v})$ if and only if there exists a differentiable function $\xi_{\mathbf{v}}^{\mathbf{x}} : [0, t_s] \rightarrow \mathbb{R}^n$ such that $\dot{\xi}_{\mathbf{v}}^{\mathbf{x}}(t) = f(\xi_{\mathbf{v}}^{\mathbf{x}}(t), \mathbf{v})$ for all $t \in [0, t_s]$, $\xi_{\mathbf{v}}^{\mathbf{x}}(0) = \mathbf{x}$, and $\xi_{\mathbf{v}}^{\mathbf{x}}(t_s) = \mathbf{x}'$. There exists a plant transition from $\mathbf{x}$ to $\mathbf{x}'$ when the controller variables are equal to $\mathbf{v}$, denoted by $\mathbf{x} \xrightarrow{F(\mathbf{v})} \mathbf{x}'$, if and only if $(\mathbf{x}, \mathbf{x}') \in F(\mathbf{v})$.

There exists a *system-level transition* between two system states $s = (L, \mathbf{v}, \mathbf{x})$ and $s' = (L', \mathbf{v}', \mathbf{x}')$, denoted by $s \Longrightarrow s'$, if and only if either:

- $(L, \mathbf{v}) \xrightarrow{\delta(\mathbf{x})} (L', \mathbf{v}')$ and $\mathbf{x} = \mathbf{x}'$; or
- $\mathbf{x} \xrightarrow{F(\mathbf{v})} \mathbf{x}'$, $L = L_{\text{final}}$, $L' = L_{\text{init}}$, and $\mathbf{v} = \mathbf{v}'$.

If the former holds, we call $s \Longrightarrow s'$ a *system-level controller transition*; if the latter holds, we call $s \Longrightarrow s'$ a *system-level plant transition*.

A *trace* of an SDCS is a finite sequence of system states $\sigma = s_0 \ldots s_K$, such that $s_0 \in Init$, and, for every $0 \leq k < K$, there exists a system-level transition $s_k \Longrightarrow s_{k+1}$. Given a trace σ, $duration(\sigma)$ denotes the amount of time elapsed between the first state and the last state of σ, and it is defined inductively as follows:

- For a trace $\sigma = s_0$, $duration(\sigma) = 0$.
- For a trace $\sigma = s_0 \ldots s_K$ such that $s_{K-1} \Longrightarrow s_K$ is a system-level controller transition, $duration(s_0 \ldots s_K) = duration(s_0 \ldots s_{K-1})$, since we assume the controller transitions execute instantaneously.
- For a trace $\sigma = s_0 \ldots s_K$ such that $s_{K-1} \Longrightarrow s_K$ is a system-level plant transition, $duration(s_0 \ldots s_K) = duration(s_0 \ldots s_{K-1}) + t_s$.

The system state $s \in S$ of an SDCS is *reachable within a time bound* T if and only if there exists a trace $\sigma = s_0 \ldots s_K$ such that $s_K = s$ and $duration(\sigma) \leq T$. A system state s

is a *deadlock state* if there does not exist a system state s' such that $s \Longrightarrow s'$. A deadlock state is a state from which no transition is possible. An SDCS has a deadlock within time bound T if and only if a deadlock state is reachable within T. A state $s = (L, \mathbf{v}, \mathbf{x})$ is a *livelock state* if and only if there exists an infinite sequence of controller states, $q_0 q_1 \ldots$, such that $q_0 = (L, \mathbf{v})$ and, for every $i \geq 0$, $q_i \xrightarrow{\delta(\mathbf{x})} q_{i+1}$. A livelock corresponds to an infinite loop in the controller. An SDCS has a livelock within time bound T if and only if there exists a livelock state that is reachable within T.

Given a set of unsafe system states $U \subset S$, an SDCS is *safe within time bound* T if and only if there exists no unsafe system state $s \in U$ that is reachable within T and the system has no deadlock or livelock within time bound T.

III. Analysis Technique

This section presents a technique for checking bounded-time safety of an SDCS. The approach, called *systematic simulation*, uses a model checker to guide the search for counterexamples. The algorithm efficiently analyzes simulation traces to determine if a system can reach an unsafe state, a deadlock, or a livelock within a given time bound.

A. Systematic Simulation

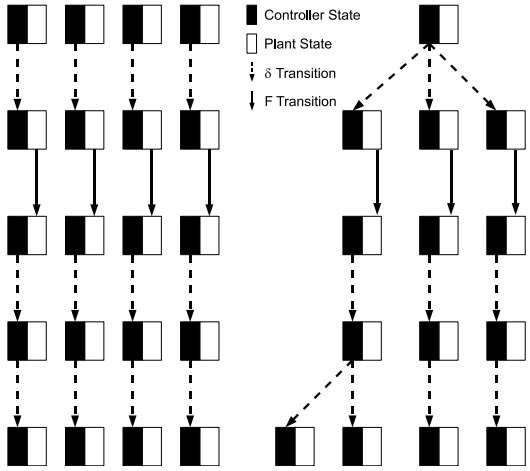
Simulation is a validation technique that generates the traces of a system. For a continuous system specified as a set of differential equations, numerical methods are used to generate traces of the system. Tools such as MATLAB/Simulink [2] are used for modeling and simulating dynamical systems. A simulation trace corresponds to one possible evolution of the dynamical system: all inputs must be fixed, and therefore the simulation is deterministic.

Model checking is a verification technique that is able to check that all possible behaviors of a system satisfy a given property. In this context, a system is allowed to be non-deterministic. Systems modeled as an SDCS exhibit non-deterministic behavior due to the following: *(i)* the interleaving of concurrent tasks; *(ii)* multiple initial states; and *(iii)* non-determinism in the controller finite state automaton, which can be used to model external inputs to the controller.

In contrast to simulation, where each simulation is independent, in model checking the set of generated traces forms a graph (see Figure 2). This leads to a saving in terms of simulation time, as simulation traces that share a common prefix are not executed twice.

The algorithm used by our approach is shown in Figure 3 and Figure 4. The main function (Figure 3) takes as inputs an SDCS, a set of unsafe states U, and a time bound T. It can return four possible answers:

- **SAFE** if the system is safe within time bound T and no deadlock or livelock is reachable within time bound T;
- **(UNSAFE, path)** if the system is unsafe; $path$ is a trace that leads to an unsafe state;
- **(DEADLOCK, path)** if a deadlock is reachable within T; $path$ is a trace that leads to a deadlock state;



(a) Standard simulation (b) Systematic simulation

Fig. 2. Standard simulation (a) generates traces one at a time. Systematic simulation (b) exploits the common prefixes of traces to make the analysis more efficient.

- **(LIVELOCK, path)** if a livelock is reachable within T; $path$ is a trace that leads to a livelock state.

The main function calls the function `explore` for each initial state of SDCS and appropriately interprets the result. The function `explore` (Figure 4) takes as arguments the system state $state$, the time horizon τ, and the sequence of states $path$. This function performs a depth-first search of the graph reachable from $state$ up to time τ.

The function `explore` first adds the current state to the sequence $path$ (line 14) and then checks if an unsafe state has been reached (line 16), in which case it returns immediately. Otherwise, `explore` checks if the current state is a livelock state (lines 18-23). Next, the current state and time horizon are compared with the set of already visited states (line 25), which is stored in the global variable `visited`: if there exists a state in `visited` that is equal to the current one with a larger or equal time horizon the search continues with a different branch; otherwise, the current state is added to the set of visited states (line 27).

Lines 30-34 correspond to a system-level plant transition and are executed if the current location, $state.L$, is equal to L_{final}. First, `explore` checks that the time horizon is large enough to allow a plant transition, whose duration is equal to the sampling time t_s. Line 32 represents the invocation of the numerical simulation procedure using $state.\mathbf{x}$ as the initial state and $state.\mathbf{v}$ as the value of the inputs. Line 34 continues the exploration by invoking `explore` recursively starting from the next state and with a shorter time horizon.

Lines 37-43 correspond to a system-level controller transition. The set of possible successors is computed at line 37. If no discrete transition is possible (line 38) a deadlock is reported. Otherwise, the successors are explored, one at a time, by the loop at lines 39-42. In this case, the time horizon is unchanged during the recursive call (line 42).

The pseudocode in Figures 3 and 4 is based on the algorithm for explicit-state model checking [8], which uses

```
1:  // Check the time-bound safety of an SDCS
2:  global SDCS; // A sampled-data control system.
3:  global U; // A set of unsafe states.
4:  global T; // A time bound.
5:  global visited ← {}; // Visited states, initially empty.
6:  main:
7:    // Perform a depth-first search for each initial state
8:    foreach (state ∈ Init)
9:      result ← explore(state, T, ());
10:     if(result ≠ SAFE_BRANCH) return result;
11:   return SAFE;
```

Fig. 3. The `main` procedure of the systematic simulation algorithm.

a depth-first search of the state transition graph. We have implemented this algorithm by modifying the search procedure of an existing model checker. The major additions are *livelock detection* (lines 18-23), storing of *time horizons* together with states in the visited set (lines 25-27), and the computation of *plant transitions* using numerical simulation (lines 30-34).

B. Approximate Equivalence

The systematic simulation approach presented in the previous section exhaustively explores all possible behaviors of an SDCS. By using a model checker, the technique is more efficient than using standard simulation to enumerate all traces. The approach, however, requires substantial computational resources when applied to complex systems. For such systems, the number of traces is exponential in the time bound and the number of tasks and inputs. The model checker has to explore all traces, even if many of them are similar to each other. In this section, we present an approach that prunes the simulation graph by ignoring some of the traces that are similar. This is a heuristic approach, which, unlike the previous algorithm, can possibly fail to detect unsafe behaviors in an SDCS, but is useful in finding errors in large systems.

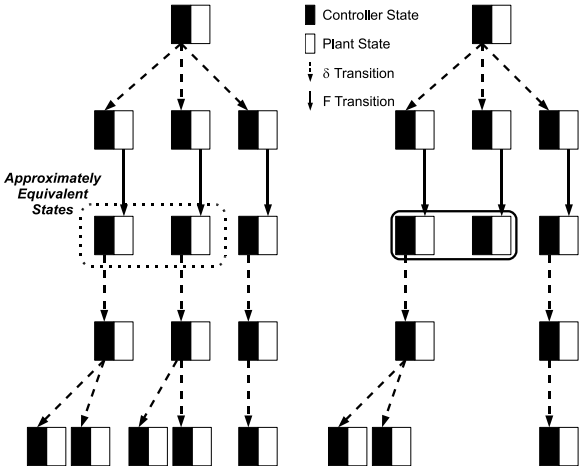
Explicit-state model checkers compute the set of reachable states iteratively by constructing a graph using the transition relation of the model. When the model checker encounters a state that is equal to a state that has already been visited (line 25 in Figure 4), the transitions starting from that state are not explored. Doing so would only lead to states that have already been encountered. In this section we present an approach that replaces the notion of state equality in model checking with state equivalence based on an approximation of the plant state. This is a heuristic approach that enables the technique to analyze larger systems. While the approach is able to efficiently search for counterexamples, it does not explore all possible system behaviors. As such it can show that the system is unsafe, but it cannot prove that a system is safe.

We introduce the notion of *approximate equivalence* for an SDCS. Two system states, $s = (L, \mathbf{v}, \mathbf{x})$ and $s' = (L', \mathbf{v}', \mathbf{x}')$, are approximately equivalent when $L = L'$, $\mathbf{v} = \mathbf{v}'$, and $\mathbf{x}$ and $\mathbf{x}'$ satisfy a proximity criterion based on their positions in $\mathbb{R}^n$. In our implementation, a grid is applied to the state space of the plant; the criterion on the positions of $\mathbf{x}$ and $\mathbf{x}'$ that we use is that both $\mathbf{x}$ and $\mathbf{x}'$ should occupy the same grid element. When our algorithm reaches a state that is approximately equivalent to a previously visited one, the transitions starting from that state are not explored. We call this operation *path pruning* (see Figure 5).

The heuristic approach presented here is not sound in that it is not guaranteed to find a counterexample if one exists. This is because the traces neglected may lead to an unsafe system state: while two plant states may be close to each other at a given point in time, they may lead to states that are far from each other. This behavior is characteristic of differential equations, where apparently simple dynamics may lead to chaotic behavior. We are currently working on an approach that has similar advantages to approximate

```
12: // Perform a depth-first search starting at state
13: function explore(state, τ, path)
14:   path ← path · state;
15:   // Check for unsafe states
16:   if (s ∈ U) return (UNSAFE, path);
17:   // Detect livelocks
18:   if (∃ N < path.length:
19:       path[N] = state ∧
20:       ∀ N ≤ k < path.length:
21:         (path[k].q, path[k+1].q) ∈ δ ∧
22:         path[k].x = path[k+1].x)
23:     return (LIVELOCK, path);
24:   // Compare to already visited states
25:   if (∃ (s,t) ∈ visited: s = state ∧ t ≥ τ)
26:     return SAFE_BRANCH;
27:   visited ← visited ∪ {(state, τ)};
28:   // Perform a plant transition
29:   if (state.L = L_final)
30:     // Stop if time horizon is less than sampling time
31:     if (τ < t_s) return SAFE_BRANCH;
32:     state.x ← F(state.x, state.v);
33:     state.L ← L_init;
34:     return explore(state, τ - t_s, path);
35:   // Perform a controller transition
36:   else
37:     succs ← {q' | (state.q, q') ∈ δ(state.x)};
38:     if (succs = ∅) return (DEADLOCK, path);
39:     foreach (q' ∈ succs)
40:       state.q ← q';
41:       result ← explore(state, τ, path);
42:       if (result ≠ SAFE_BRANCH) return result;
43:     return SAFE_BRANCH;
```

Fig. 4. The `explore` function.



(a) Approximate equivalence (b) Path pruning

Fig. 5. Approximate equivalence (a) identifies states that have the same controller state and similar plant states. This enables pruning (b) parts of the graph.

equivalence but is able to prove bounded-time safety of an SDCS.

For stable affine plant dynamics, we can determine a Lyapunov function, $\mathcal{V}(\mathbf{x}) = \mathbf{x}^T\mathbf{P}\mathbf{x}$, where $\mathbf{P} \in \mathbb{R}^{n\times n}$ is positive definite. We define a Lyapunov ellipsoid with center $\mathbf{x}_c$ and size $\alpha \geq 0$ as $\mathcal{E}(\mathbf{x}_c, \mathbf{P}, \alpha) = \{\mathbf{x} | (\mathbf{x} - \mathbf{x}_c)^T\mathbf{P}(\mathbf{x} - \mathbf{x}_c) \leq \alpha\}$, where the matrix $\mathbf{P}$ determines the shape. Lyapunov ellipsoids have the following property. Given two plant states $\mathbf{x}$ and $\mathbf{y}$ such that $\mathbf{y} \in \mathcal{E}(\mathbf{x}, \mathbf{P}, \alpha)$, if $\mathbf{x} \xrightarrow{F(\mathbf{v})} \mathbf{x}'$ and $\mathbf{y} \xrightarrow{F(\mathbf{v})} \mathbf{y}'$, then $\mathbf{y}' \in \mathcal{E}(\mathbf{x}', \mathbf{P}, \alpha)$. Consequently, if we know that $\mathbf{x}$ and $\mathbf{y}$ are sufficiently close, meaning if $\|\mathbf{x}-\mathbf{y}\|_P \leq \alpha$ for a given $\alpha > 0$, then we know that $\mathbf{x}'$ and $\mathbf{y}'$ remain sufficiently close, that is $\|\mathbf{x}' - \mathbf{y}'\|_P \leq \alpha$ (see Figure 6). We will use this property of Lyapunov ellipsoids to define an equivalence relation which preserves bounded-time safety.

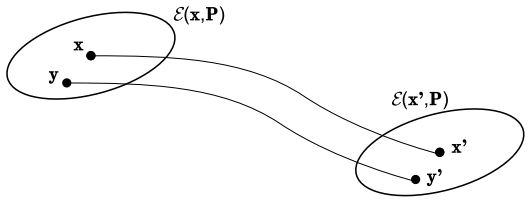


Fig. 6. If $\mathbf{y}$ is within a Lyapunov ellipsoid of size α centered at $\mathbf{x}$, and $\mathbf{x}'$ and $\mathbf{y}'$ are plant states reachable from $\mathbf{x}$ and $\mathbf{y}$, then $\mathbf{y}'$ is within a Lyapunov ellipsoid of size α centered at $\mathbf{x}'$.

IV. Experimental Evaluation

We implemented our technique by extending an existing explicit-state source code model checker. The tool we chose is Java PathFinder [15]. While the main purpose of the tool is to verify Java programs, it is able to handle the subset of C that is common to the two languages. We were able to check the code automatically generated using the MathWorks' Real-Time Workshop with only minor syntactic modifications. The main reasons for choosing this tool were that it was readily available and it could be extended to implement our approach. In future work, we plan to investigate using alternative model checkers, especially tools that are aimed at C/C++. We used MATLAB/Simulink to model the plant and controller and to provide simulation traces for the systematic simulation analysis.

We extended the existing model checker in the following way. We added an additional component to the state of the system corresponding to the state of the plant. The plant state is represented by a set of floating-point values for each of the continuous state variables. We extended the transition system constructed by the model checker to include plant transitions.

Separate concurrent processes are modeled explicitly in the model checker; the model checker automatically transforms the separate tasks into a single nondeterministic transition system. Plant transitions are computed using the MATLAB/Simulink numerical integration solver (this corresponds to line 32 in Figure 4). Given the sampling period t_s, the current plant state $\mathbf{x}$, and the value of the program state $\mathbf{v}$, MATLAB/Simulink returns the state $\mathbf{x}'$ that is reached at time t_s.

In the following, we present experimental results we obtained by applying our technique to an example based on the Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC). STARMAC is a quadrotor unmanned aerial vehicle (UAV) under development at Stanford University [11]. We obtained a Simulink model of the STARMAC system from the Stanford development team. We believe that the model is correct and does not contain an error. We constructed a new system model, the Reconnaissance Mission (RM) model, that includes a supervisory controller that we designed. We used our technique to detect an error in the RM model.

The vehicle, shown in Figure 7, is composed of a computer controller and power supply at its center, which is attached to a frame on which four rotors are mounted. The controller of the vehicle is organized on three levels illustrated in Figure 8. The inner loop controller sends thrust commands to the four rotors based on the pitch, roll, yaw, and altitude commands that it receives from the outer loop controller. The latter commands are based on the position command (in three dimensions) that the supervisory controller generates. The supervisor makes its decision based on the current position of the vehicle.

We constructed a supervisory controller whose purpose is to guide the vehicle through a sequence of waypoints.[1] The controller must be robust with respect to invalid waypoints, meaning that it has to guarantee that the vehicle will not reach an altitude below 1 meter unless it is taking off or landing (corresponding to the first and last waypoint in the sequence). The supervisory controller is modeled using

[1]Note that this controller was implemented to demonstrate the ability of the model checker to find error conditions. It is not the original supervisory controller designed for the STARMAC.

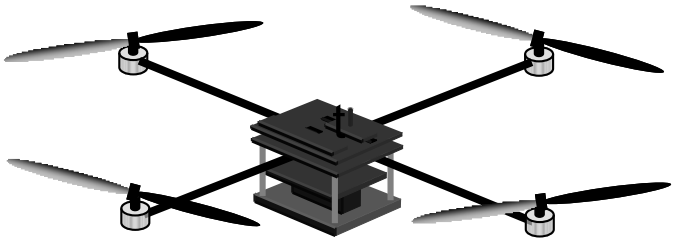


Fig. 7. An illustration of the Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control.

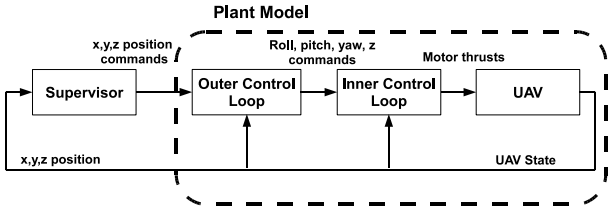


Fig. 8. Vehicle block diagram.

Stateflow diagrams. The implementation uses the following interleaved tasks, illustrated in Figure 9:

- **Waypoint Tracking** - takes the vehicle through a set of positions given by a waypoint list. It checks the proximity of the vehicle to the target waypoint and, if the vehicle is close to the target, it then picks the next waypoint from the list and issues the command to the STARMAC Quadrotor.
- **Waypoint Monitor** - checks if the altitude command of the next waypoint is below 1.1 meters and, if so, it adjusts the altitude command to 1.1 meters.
- **Command Latch** - maintains the last command until the next waypoint command is issued.

The tasks communicate among themselves using shared variables.

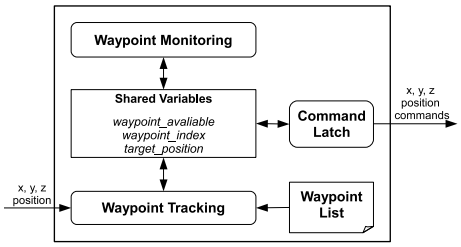


Fig. 9. Block diagram of the supervisory controller.

The MATLAB/Simulink model of the RM system includes the supervisory controller, the outer loop controller, the inner loop controller, and the dynamics of the vehicle (see Figure 8). Since the inner and outer control loops operate at a much faster clock rate than the supervisor, we model them as part of the RM plant and take the supervisor to be the RM controller.

The RM plant model corresponds to a set of non-linear differential equations with over 39 continuous-valued state variables. The interaction between the plant and the supervisory controller occurs by means of position commands (in the x, y, and z coordinates) sent by the supervisor to the plant, and position sensor values sent by the plant to the supervisor.

The property that we want to check is that the vehicle never flies below the minimum safe altitude of 1 meter, unless it is taking off or landing.

We used the technique described in Section III to search for a counterexample. The tool explores the traces of the system until it reaches an unsafe state and the counterexample shown in Figure 10 is generated. The horizontal axis in the figure represents time, the vertical one represents the altitude. The dashed curve is the actual altitude of the vehicle as it evolves with the passing of time. The solid curve represents the altitude command generated by the controller. The trace is a counterexample because at the end of the trace the altitude reaches a value below 1 meter and the vehicle is neither taking off nor landing. The circles on the diagram mark the sampling times and may correspond to multiple controller transitions.

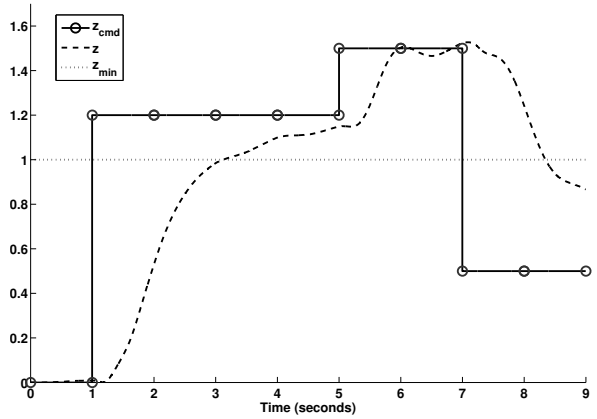


Fig. 10. Counterexample trace.

The counterexample is due to the interleaving of the tasks. In this particular trace the Waypoint Monitor task executes before the Waypoint Tracking task at time $t = 7$ seconds and therefore sees the previous value of `target_position`. Since this value is valid (its altitude component is above 1.1 meters) the value is not changed. After that, the Waypoint Tracking task executes and `target_position` is set equal to the fourth waypoint, which contains an invalid altitude value (see Figure 11). The value of `waypoint_available` is set to true and the Command Latch task records the incorrect value. At this point the vehicle starts to decrease its altitude towards the waypoint at altitude 0.5 meters. At the next sample time, the Waypoint Monitor task corrects the value, but it is too late as `waypoint_available` is now set to false and the Command Latch task does not update its interval value until the next waypoint is generated. One sampling time later the vehicle altitude becomes lower than the minimum safe altitude and an error is reported by our tool.

#	x	y	z
1	0.0	0.0	0.0
2	2.0	1.3	1.2
3	0.2	2.0	1.5
4	1.8	1.1	0.5
5	1.2	0.4	1.5
6	0.0	0.0	0.0

Fig. 11. List of waypoints used in the experimental evaluation

As shown in Figure 12, during the analysis with a time bound of 15 seconds, the tool generated 131,158 states before detecting the error. This required about 11 minutes and 928MB of memory. The counterexample shown in Figure 10 contains 1346 transitions, of which 9 are plant transitions and the rest are controller transitions. The large number of controller transitions is due to the fact that the software is modeled at the statement level in order to be able to check the interleaving of the tasks. During the analysis, the tool encountered 140,673 states equivalent to previously visited states, marked as revisited states in the table. In the example we analyzed most of the revisited states were actually identical to previously visited states. We believe this is due to the fact that most of the revisited states are obtained by a different interleaving of the tasks: different task orderings during execution often led to the same state. The approach, however, is able to detect those cases where a different ordering leads to a different behavior, as in the counterexample shown above. The results for different values of the time bound are shown in Figure 12. Notice that no counterexample is found for a time bound of 5 seconds (first row in the table), since the duration of the shortest counterexample trace is 9 seconds.

Time bound	Running time	Memory usage	Reached states	Revisited states
5s[1]	5:50s	795MB	112,057	131,781
10s	1:12s	25MB	2,470	2,449
15s	11:31s	928MB	131,158	140,673

[1]No counterexample found.

Fig. 12. Running times, memory usage, number of reached states, and number of revisited states for different time bounds and with and without approximate equivalence.

V. Conclusions

This paper presents an approach for the validation of sampled-data control systems where the controller is implemented as a set of concurrent tasks and the plant is described by a set of differential equations. We have implemented our approach using an explicit-state source code model checker for handling the controller and MATLAB/Simulink for simulating models of the plant.

Our approach uses numerical simulation in conjunction with a model checker to provide an efficient way to explore a large set of system behaviors and detect possible errors. One of the main advantages of this approach is that it can accurately model the controller and is able to find errors that are difficult to identify using simulation. By using a source code model checker and MATLAB/Simulink, we believe these techniques can be used to address industrial problems.

As described in Section III, we are currently working on a technique based on ellipsoids and Lyapunov functions to strengthen the results that can be obtained using our technique. We are also implementing the method using a model checker that handles C/C++ directly in order to increase the applicability of our approach.

References

[1] *Simulink Design Verifier User's Guide*. The MathWorks, 2007.

[2] *Using Simulink*. The MathWorks, 2007.

[3] Thomas Ball and Sriram K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *Proc. of the 7th International SPIN Workshop*, 2000.

[4] Oleg Botchkarev and Stavros Tripakis. Verification of Hybrid Systems with Linear Differential Inclusions Using Ellipsoidal Approximations. In *Proc. of the 3rd International Workshop on Hybrid Systems: Computation and Control*. Springer-Verlag, 2000.

[5] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular Verification of Software Components in C. In *Proc. of the 25th International Conference on Software Engineering*, 2003.

[6] Alongkrit Chutinan and Bruce H. Krogh. Verification of Infinite State Dynamic Systems Using Approximate Quotient Transition Systems. *IEEE Transactions on Automatic Control*, 46(9):1401–1410, 2001.

[7] Edmund M. Clarke and E. Allen Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Proc. of Workshop on Logic of Programs*, 1981.

[8] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[9] Thomas A. Henzinger. The Theory of Hybrid Automata. In *Proc. of the 11th Annual IEEE Symposium on Logic in Computer Science*, 1996.

[10] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy Abstraction. In *Proc. of the 29th Symposium on Principles of Programming Languages*, 2002.

[11] Gabriel M. Hoffmann, Haomiao Huang, Steven L. Waslander, and Claire J. Tomlin. Quadrotor Helicopter Flight Dynamics and Control: Theory and Experiment. In *Proc. of the AIAA Guidance, Navigation, and Control Conference*, 2007.

[12] Stefan Kowalewski, Sebastian Engell, Jörg Preußig, and Olaf Stursberg. Verification of Logic Controllers for Continuous Plants Using Timed Condition/Event-System Models. *Automatica*, 35(3):505–518, 1999.

[13] Klaus D. Müller-Glaser, Gerd Frick, Eric Sax, and Markus Kühl. Multiparadigm Modeling in Embedded Systems Design. *IEEE Transactions on Control Systems Technology*, 12(2):279–292, March 2004.

[14] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, 1982.

[15] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.