Verification of Supervisory Control Software Using State Proximity and Merging*

Flavio Lerda[1]**, James Kapinski[2], Edmund M. Clarke[1], and Bruce H. Krogh[2]

[1] School of Computer Science
flerda@cs.cmu.edu, emc@cs.cmu.edu
[2] Department of Electrical and Computer Engineering
jpk3@ece.cmu.edu, krogh@ece.cmu.edu
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract. This paper describes an approach for bounded-time verification of safety properties of supervisory control software interacting with a continuous-time plant. A combination of software Model Checking and numerical simulation is used to compute a conservative approximation of the reachable states. The technique verifies system properties in the presence of nondeterministic behavior in the software due to, for instance, interleaving of tasks. A notion of *program equivalence* is used to characterize the behaviors of the controller, and the bisimulation functions of Girard and Pappas are employed to characterize the behaviors of the plant. These notions are used to compute sets of plant states around a trace that are guaranteed to be safe. These sets are determined by a backward analysis that starts from the end of a trace and propagates the safe sets towards the initial states. By using these safe sets, the approach can conservatively merge traces that reach states that are in proximity to each other. The technique has been implemented for the case of affine plant dynamics, which allow efficient operations on ellipsoidal sets based on convex optimizations involving linear matrix inequalities (LMIs). We present an illustrative example for a model of the position controller of an unmanned aerial vehicle (UAV).

1 Introduction

Model-based design of embedded control systems is becoming standard practice. Applying formal methods to embedded control design is important for reducing time to market and for meeting safety and performance requirements, but formal methods are difficult to apply to systems that interact with a continuous dynamic environment. We present a formal verification technique based

* This research was sponsored by the Air Force Research Office (AFRO) under contract no. FA9550-06-1-0312, and by the National Science Foundation (NSF) under grant no. CCR-0411152. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of AFRO, NSF, or the U.S. government.

** The first author was supported by General Motors under grant no. GM9100096UMA.

on the combination of software Model Checking and numerical simulation of a continuous dynamic plant. We use level sets of bisimulation functions [6] for the plant and a notion of *program equivalence* for the controller to guarantee safety bounds and provide an efficient and exhaustive search of the system behaviors. The approach narrows the gap between simulation and Model Checking of control systems.

A nonconservative approach that combines Model Checking [3, 4] and simulation was first proposed in [13]. That approach provides a means of efficiently searching for counterexamples, but since it is not conservative, it cannot guarantee safety. The approach presented here formalizes and extends that technique by employing conservative approximations of the set of reachable states.

Various methods have been developed to formally verify hybrid automata, which can model embedded control systems [8]. These techniques are computationally expensive, however, and are able to analyze only systems of low complexity. Reachable set estimation is a central problem in performing verification of safety properties. Other techniques perform a search of the reachable set of states forward in time and merge the reachable trajectories that are in proximity to each other in the state space [11, 5]. The work here goes further by using the safety requirements to construct sets of states that are guaranteed to be safe.

The work by Julius et al. provides a means for determining maximum safety bounds for simulation traces [10], but the technique does not handle nondeterminism in the discrete transitions and it does not consider the semantics of the control software. The work presented here deals efficiently with the proliferation of reachable paths that occur due to nondeterministic behaviors in the controller.

The paper is organized as follows. Section 2 describes the model of a supervisory controller interacting with a continuous-time plant. Section 3 presents the theoretical foundation for our technique and the bounded-time verification algorithm. Section 4 describes the implementation of our technique and presents experimental results for an example that is based on a UAV system. Conclusions and directions for further work are presented in Section 5.

2 System Model

This paper concerns verification of *supervisory controllers*, by which we mean feedback controllers that select operating modes for continuous dynamic systems. The supervisor may select plant operating modes directly or manage lower-level feedback control loops. For the purposes of our analysis, lower-level control loops are modeled as part of the plant. This is appropriate if the lower level controller has a significantly higher sampling rate than the supervisor. A sampled-data supervisor is able to observe the state of the plant only at fixed times, called sample instants. We assume that the sample instants are multiples of a fixed sampling period, $t_s > 0$. In this section we present a model of the system where the supervisor is represented by a set of tasks, and the plant is described by a set of differential equations. We assume that the code of the supervisor executes instantaneously, which is a reasonable assumption if the sampling period of the

supervisor is large compared to the actual execution time of the code. Also, we assume that all tasks share the same clock. This assumption is appropriate for analyzing control software implemented as multiple tasks running on one processor. Systems implemented as tasks running on multiple processors can also be analyzed using this technique if the clock skew and jitter are small relative to the sampling period of the tasks.

Consider a set of m supervisor variables taking values from a finite set V, and a set of n real-valued plant variables. Let $\mathbf{v} \in V^m$ be the value of the supervisor variables, and $\mathbf{x} \in \mathbb{R}^n$ be the value of the plant variables, called the plant state.

Definition 1 (Supervisor Task). *Given a set of supervisor variables V^m and a set of plant states $\mathbb{R}^n$, a supervisor task is a tuple $\mathcal{T}_i = \langle Loc_i, l_{i,initial}, l_{i,final}, \delta_i \rangle$ where:*

- *Loc_i is a finite set of control locations;*
- *$l_{i,initial}, l_{i,final} \in Loc_i$ are two specially designated locations, called the initial and final control locations of $\mathcal{T}_i$; and*
- *$\delta_i : \mathbb{R}^n \rightarrow 2^{Loc_i \times V^m \times Loc_i \times V^m}$ is the transition relation of $\mathcal{T}_i$. We assume that there are no transitions from the final control location $l_{i,final}$.*

At each sample instant, the task starts executing at the initial control location $l_{i,initial}$ and executes until it reaches the final control location $l_{i,final}$. Let us assume that every sequence of task transitions is finite and eventually reaches the control location $l_{i,final}$, i.e., the code has no deadlock or livelock. A Model Checker can be used to detect deadlocks and livelocks, but these aspects have been omitted from the presentation for the sake of clarity. An approach that takes into account these aspects is described in [13]. Notice that the transition relation δ_i depends on the current plant state $\mathbf{x}$. Given $l_i, \hat{l}_i \in Loc_i$, $\mathbf{v}, \hat{\mathbf{v}} \in V^m$, and $\mathbf{x} \in \mathbb{R}^n$, there exists a transition from $(l_i, \mathbf{v})$ to $(\hat{l}_i, \hat{\mathbf{v}})$ when the plant state is equal to $\mathbf{x}$ if and only if $(l_i, \mathbf{v}, \hat{l}_i, \hat{\mathbf{v}}) \in \delta_i(\mathbf{x})$.

Definition 2 (Sampled-Data Control System). *A sampled-data control system is a tuple $SDCS = \langle \{\mathcal{T}_1, \ldots, \mathcal{T}_p\}, V, f_{\mathbf{v}}, t_s, Init \rangle$ where:*

- *$\{\mathcal{T}_1, \ldots, \mathcal{T}_p\}$ is a finite set of supervisor tasks;*
- *V is a finite domain for the supervisor variables;*
- *For each $\mathbf{v} \in V^m$, $f_{\mathbf{v}} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is a Lipschitz continuous function that describes the flow of the plant and depends on the value of the supervisor variables;*
- *t_s is the sampling period; and*
- *$Init \subseteq Loc_1 \times \ldots \times Loc_p \times V^m \times \mathbb{R}^n$ is a set of initial states.*

Let Loc denote the set $Loc_1 \times \ldots \times Loc_p$ of the control locations of each task. A state of an *SDCS* is a tuple $(q, \mathbf{x})$ where: $q = (L, \mathbf{v})$ is the supervisor state, $L \in Loc$ specifies the control locations of each task, $\mathbf{v} \in V^m$ is the value of the supervisor variables, and $\mathbf{x} \in \mathbb{R}^n$ is the plant state. Given a value $\mathbf{v}$ for the supervisor variables and a plant state $\mathbf{y}$, let $\xi^{\mathbf{y}}_{\mathbf{v}} : \mathbb{R} \rightarrow \mathbb{R}^n$ denote a solution to the initial value problem $\dot{\mathbf{x}}(t) = f_{\mathbf{v}}(\mathbf{x}(t)), \mathbf{x}(0) = \mathbf{y}$. Since we assumed that $f_{\mathbf{v}}(\cdot)$ is Lipschitz continuous, there exists a unique $\xi^{\mathbf{y}}_{\mathbf{v}}(\cdot)$ for every $\mathbf{y} \in \mathbb{R}^n$.

Definition 3 (Transitions). *Given two states $s = (q, \mathbf{x})$ and $\hat{s} = (\hat{q}, \hat{\mathbf{x}})$ of an SDCS, there exists a transition from s to $\hat{s}$, denoted by $s \longrightarrow \hat{s}$, if either:*

- $q = ((l_1, \ldots, l_p), \mathbf{v})$, $\hat{q} = ((\hat{l}_1, \ldots, \hat{l}_p), \hat{\mathbf{v}})$, *and there exists a task $\mathcal{T}_j$ such that $\mathbf{x} = \hat{\mathbf{x}}$, $(l_j, \mathbf{v}, \hat{l}_j, \hat{\mathbf{v}}) \in \delta_j(\mathbf{x})$ and, for every task T_i not equal to T_j, $l_i = \hat{l}_i$. This is called a supervisor transition.*
- $q = (L_{final}, \mathbf{v})$, $\hat{q} = (L_{initial}, \mathbf{v})$, *and $\hat{\mathbf{x}} = \xi_{\mathbf{v}}^{\mathbf{x}}(t_s)$. This is called a plant transition.*

A trace of an *SDCS* is a finite sequence of states $\sigma = s_0 \ldots s_K$, for some K, such that $s_k \longrightarrow s_{k+1}$ for all $0 \leq k < K$. Figure 1 provides an illustration of traces of an SDCS. The initial state is $s_0 = (L_{final}, \mathbf{v}_a, \mathbf{x}_{init})$, and the first transition is a plant transition, $s_0 \longrightarrow s_1$, where $s_1 = (L_{initial}, \mathbf{v}_a, \hat{\mathbf{x}})$ for some $\hat{\mathbf{x}}$. Several supervisor transitions are possible starting from s_1; nondeterminism in the supervisor leads to two separate states, $s_2 = (L_{final}, \mathbf{v}_b, \hat{\mathbf{x}})$ and $s_4 = (L_{final}, \mathbf{v}_c, \hat{\mathbf{x}})$. From each of these states a plant transition is taken, $s_2 \longrightarrow s_3$, where $s_3 = (L_{initial}, \mathbf{v}_b, \mathbf{y})$ for some $\mathbf{y}$, and $s_4 \longrightarrow s_5$, where $s_5 = (L_{initial}, \mathbf{v}_c, \mathbf{z})$ for some $\mathbf{z}$.

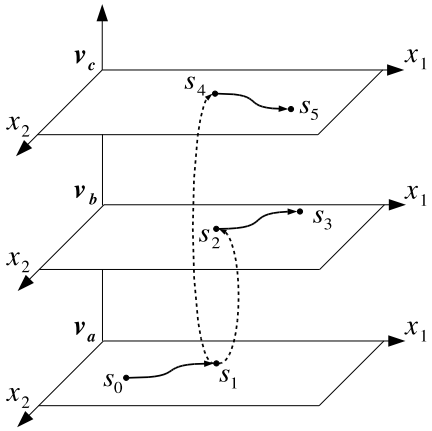

Fig. 1. An illustration of traces of an SDCS. Solid arrows connecting points represent plant transitions. Dotted lines connecting points represent sequences of supervisor transitions.

Definition 4 (Duration). *The duration of a trace σ is the amount of time elapsed between its first state and its last state, and it is defined inductively as follows:*

- *If $\sigma = s_0$, $duration(\sigma) = 0$.*

- *If* $\sigma = s_0 \dots s_K$ *and* $s_{K-1} \longrightarrow s_K$ *is a supervisor transition then* $duration(\sigma) = duration(s_0, \dots s_{K-1})$, *since we assume that supervisor transitions execute instantaneously.*
- *If* $\sigma = s_0 \dots s_K$ *and* $s_{K-1} \longrightarrow s_K$ *is a plant transition then* $duration(\sigma) = duration(s_0, \dots s_{K-1}) + t_s$.

A state s of an *SDCS* is reachable within a time bound T if and only if there exists a trace $\sigma = s_0 \dots s_K$, for some K, such that $s_0 \in Init$, $s_K = s$ and $duration(\sigma) \leq T$. Given a time bound T and a set of states $Fail \subset Loc \times V^m \times \mathbb{R}^n$, a state s is safe for time bound T if and only if for every trace $\sigma = s_0 \dots s_K$, of arbitrary length K, such that $s_0 = s$ and $duration(\sigma) \leq T$, we have that $s_K \notin Fail$. For example, state s_0 in Figure 1 is safe for time bound $2t_s$.

Definition 5 (Bounded-Time Safety). *Given an SDCS, a set* $Fail \subset Loc \times V^m \times \mathbb{R}^n$ *of fail states, and a time bound* T, *the SDCS is safe for time bound* T *if and only if all initial states are safe for time bound* T.

3 Conservative Verification using Merging

In [13], we presented an approach that combines Model Checking and simulation to check bounded-time safety of an *SDCS* with a finite set of initial states. That work also introduces a notion of *approximate equivalence* that is used to prune the state space and, therefore, reduces the size of the state space that needs to be explored. The approach is not conservative, however; it can be used to search for counterexamples, but it is unable to prove safety.

Our approach for proving bounded-time safety of an *SDCS* is able to prune parts of the state space by *merging traces*, which corresponds to merging a state with a previously visited one. In Model Checking, merging can be done only when a state on one trace is identical to a state on another trace. Our approach is able to perform a merge when two states are in proximity to each other if the pruned parts of the state space are guaranteed to be safe. In the following, we show how to determine safe sets of plant states around the points in a trace. These sets correspond to a set of traces that are in proximity of the visited trace and are guaranteed to be safe. When a state that is within a safe set is reached, the trace can be merged conservatively and the successors of such a state do not need to be explored further.

In general, given a dynamical system and two initial states that are in proximity to each other, the trajectories starting at those initial states may diverge. This paper uses bisimulation functions to bound the distance between future evolutions. Bisimulation functions were introduced by Girard and Pappas as a way to determine the relation between states of a dynamical system [6]. In this work, we use bisimulation functions to approximate conservatively the plant transitions.

Definition 6 (Bisimulation Function). [6] *Given an autonomous dynamical system* Σ *described by* $\dot{\mathbf{x}}(t) = f_{\mathbf{v}}(\mathbf{x}(t))$ *where* $\mathbf{x} : \mathbb{R} \to \mathbb{R}^n$ *and* $f : \mathbb{R}^n \to \mathbb{R}^n$, *a*

differentiable function $\varphi_{\mathbf{v}} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ *is a bisimulation function of* Σ *if and only if*

- $\varphi_{\mathbf{v}}(\mathbf{y}, \mathbf{z}) \geq 0$, *for all* $\mathbf{y}, \mathbf{z} \in \mathbb{R}^n$*; and*
- $\nabla_{\mathbf{y}} \varphi_{\mathbf{v}}(\mathbf{y}, \mathbf{z}) \cdot f_{\mathbf{v}}(\mathbf{y}) + \nabla_{\mathbf{z}} \varphi_{\mathbf{v}}(\mathbf{y}, \mathbf{z}) \cdot f_{\mathbf{v}}(\mathbf{z}) \leq 0$, *for all* $\mathbf{y}, \mathbf{z} \in \mathbb{R}^n$.

Definition 7 (Sublevel Sets). *Given* $\mathbf{x} \in \mathbb{R}^n$*, a bisimulation function* φ *of* $\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$*, and a real value* $r \geq 0$*, the sublevel set of the bisimulation function* φ *centered at* $\mathbf{x}$ *and of size* r*, denoted by* $\mathcal{N}_{\varphi}(\mathbf{x}, r)$*, is defined as*

$$\mathcal{N}_{\varphi}(\mathbf{x}, r) = \{\mathbf{z} \in \mathbb{R}^n \,|\, \varphi(\mathbf{x}, \mathbf{z}) \leq r\}.$$

In the following, we assume that the bisimulation functions are symmetric, i.e., $\varphi(\mathbf{y}, \mathbf{z}) = \varphi(\mathbf{z}, \mathbf{y})$ for every $\mathbf{y}, \mathbf{z} \in \mathbb{R}^n$. If a bisimulation function $\varphi(\cdot, \cdot)$ is a metric on $\mathbb{R}^n$, then it is called a *contraction metric* [1]. We assume that for every value of the supervisor variables $\mathbf{v}$, a bisimulation function $\varphi_{\mathbf{v}}$ of the autonomous dynamical system $\dot{\mathbf{x}}(t) = f_{\mathbf{v}}(\mathbf{x}(t))$ is given. We can now state the following theorem about bisimulation functions and plant transitions, based on a theorem from Julius et al. [10].

Theorem 1 (Plant Approximation). *Given two states* $s = ((L_{\mathit{final}}, \mathbf{v}), \mathbf{y})$ *and* $\hat{s} = ((L_{\mathit{initial}}, \mathbf{v}), \hat{\mathbf{y}})$ *such that* $s \longrightarrow \hat{s}$ *is a plant transition, and a bisimulation function* $\varphi_{\mathbf{v}}$ *for the differential equation* $\dot{\mathbf{x}}(t) = f_{\mathbf{v}}(\mathbf{x}(t))$*, for every* $r \geq 0$ *and for every* $\mathbf{z} \in \mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{y}, r)$*, if* $((L_{\mathit{final}}, \mathbf{v}), \mathbf{z}) \longrightarrow ((L_{\mathit{initial}}, \mathbf{v}), \hat{\mathbf{z}})$ *is a plant transition, then* $\hat{\mathbf{z}} \in \mathcal{N}_{\varphi_{\mathbf{v}}}(\hat{\mathbf{y}}, r)$*.*

Proof. The theorem is a direct consequence of Corollary 1 of [10].

Given a program state q and a time bound T, a set $X \subseteq \mathbb{R}^n$ of plant states is safe for T at q if and only if, for every $\mathbf{x} \in X$, $s = (q, \mathbf{x})$ is safe for time bound T. Given a program state q, the set of fail plant states at q is defined as $\mathit{Fail}_q = \{\mathbf{x} \in \mathbb{R}^n \,|\, (q, \mathbf{x}) \in \mathit{Fail}\}$.

Theorem 2 (Plant Transition Approximation). *Given two states* $(q, \mathbf{y})$ *and* $(\hat{q}, \hat{\mathbf{y}})$ *such that* $q = (L_{\mathit{final}}, \mathbf{v})$*,* $\hat{q} = (L_{\mathit{initial}}, \mathbf{v})$*, and* $(q, \mathbf{y}) \longrightarrow (\hat{q}, \hat{\mathbf{y}})$ *is a plant transition, if* $\hat{\mathcal{X}} \subseteq \mathbb{R}^n$ *is safe for* T *at* $\hat{q}$*, then for all* $r \geq 0$*, if* $\mathcal{N}_{\varphi_{\mathbf{v}}}(\hat{\mathbf{y}}, r) \subseteq \hat{\mathcal{X}}$ *and* $\mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{y}, r) \subseteq \overline{\mathit{Fail}_q}$ *then* $\mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{y}, r)$ *is safe for* $(T + t_s)$ *at* q*.*

Proof. We prove this theorem by contradiction. Assume that $\mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{y}, r)$ is not safe for $(T + t_s)$ at q. This means that there exists a plant state $\mathbf{z} \in \mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{y}, r)$ and a trace $\sigma = s_0 s_1 \dots s_K$, for some K, such that $s_0 = (q, \mathbf{z})$, $s_K \in \mathit{Fail}$, and $\mathit{duration}(\sigma) \leq T + t_s$. Since $\mathbf{z} \in \mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{y}, r) \subseteq \overline{\mathit{Fail}_q}$, we have that $s_0 \notin \mathit{Fail}$ and therefore the trace must contain at least two states ($K \geq 1$). Let $\hat{\sigma}$ denote $s_1 \dots s_K$. By Definition 4 we have that $\mathit{duration}(\hat{\sigma}) = \mathit{duration}(\sigma) - t_s \leq T$. By Definition 3, $s_1 = (\hat{q}, \hat{\mathbf{z}})$ for some $\hat{\mathbf{z}} \in \mathbb{R}^n$. By Theorem 1 we can deduce that $\hat{\mathbf{z}} \in \mathcal{N}_{\varphi_{\mathbf{v}}}(\hat{\mathbf{y}}, r)$. But, by hypothesis, $\mathcal{N}_{\varphi_{\mathbf{v}}}(\hat{\mathbf{y}}, r) \subseteq \hat{\mathcal{X}}$ and therefore $\hat{\mathbf{z}} \in \hat{\mathcal{X}}$. Since $\hat{\mathcal{X}}$ is safe for T at $\hat{q}$, there does not exists any trace starting at $(\hat{q}, \hat{\mathbf{z}})$ that reaches a state in *Fail* and whose duration is less than or equal to T. However, $\hat{\sigma}$ is such a trace, which is a contradiction, therefore $\mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{y}, r)$ must be safe for $(T + t_s)$ at q. □

Theorem 2 allows us to determine a set of plant states that are safe for $(T + t_s)$ at a given supervisor state q given a set of plant states that are safe for T at the supervisor state $\hat{q}$ obtained by performing a plant transition. Below we show how to compute a set of plant states that is safe for T at a supervisor state q for the case of discrete transitions. While continuous transitions are always deterministic, supervisor transitions may lead from one state to a number of successor states. In order to deal with this, we define a notion of equivalence between continuous states with respect to a supervisor state.

Figure 2-(a) illustrates the notion of safe plant states and plant transition approximations. In the example, the plant states have two dimensions, corresponding to the axes labeled x_1 and x_2. The vertical axis represents the value of the supervisor variables: each plane corresponds to a different value of the supervisor variables, namely $\mathbf{v}_a, \mathbf{v}_b$, and $\mathbf{v}_c$. On each plane, the areas marked by *Fail* correspond to the parts of the plant state space that are unsafe for the corresponding value of the supervisor variables. Plant transitions correspond to continuous lines within a given plane; supervisor transitions correspond to dotted lines from one plane to another. The two sets N_3 and N_5 are safe for time bound zero as they do not intersect the *Fail* plant states in the corresponding planes. By Theorem 2, the sets N_2 and N_4 are safe for time bound t_s, the sampling period, as they are guaranteed to avoid the *Fail* region if the system evolves for one sampling period.

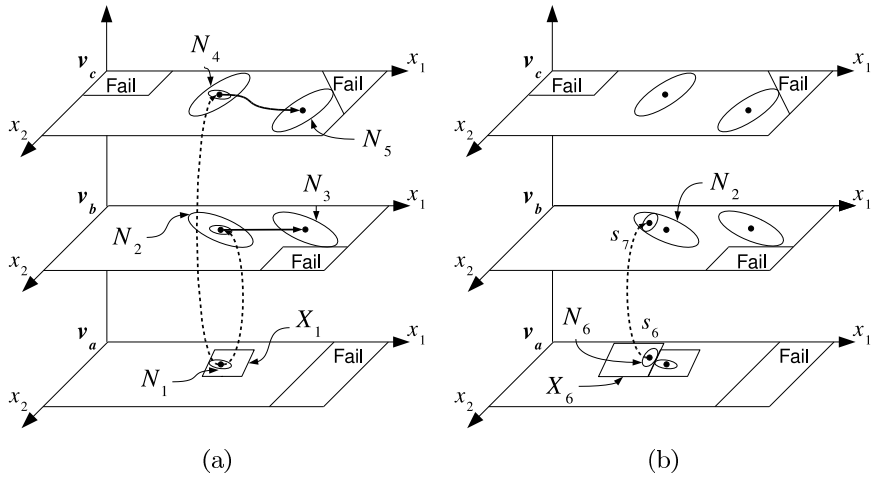

Fig. 2. (a) An illustration of sets safe for a time bound T. N_3 and N_5 are safe for time bound zero. N_1 N_2 and N_4 are safe for t_s; (b) An illustration of merging. N_6 is safe for t_s since all of its states lead within the set N_2 which is safe for t_s.

Definition 8 (Program Equivalence). *Given a supervisor state q and a pair of plant states $\mathbf{y}, \mathbf{z} \in \mathbb{R}^n$, we say that $\mathbf{y}$ is program equivalent to $\mathbf{z}$ at q, denoted by $\mathbf{y} \approx_q \mathbf{z}$, if the set of successors of q at plant state $\mathbf{y}$ is the same as the set of successors of q at plant state $\mathbf{z}$, i.e., $\hat{Q}_q(\mathbf{y}) = \hat{Q}_q(\mathbf{z})$ where, given a supervisor state q and a plant state $\mathbf{x}$, $\hat{Q}_q(\mathbf{x}) = \{\hat{q} \,|\, (q, \mathbf{x}) \longrightarrow (\hat{q}, \mathbf{x})\}$.*

The relation $\approx_q$ defined above is an equivalence relation. Therefore, for every supervisor state q, $\approx_q$ defines a set of equivalence classes. Given a supervisor state q and a plant state $\mathbf{y}$, let $[\mathbf{y}]_{\approx_q}$ denote the equivalence class of $\mathbf{y}$ defined by $\approx_q$, that is $[\mathbf{y}]_{\approx_q} = \{\mathbf{z} \in \mathbb{R}^n \,|\, \mathbf{y} \approx_q \mathbf{z}\}$.

Theorem 3 (Supervisor Transition Approximation). *Given a state $(q, \mathbf{y})$ with $q = (L, \mathbf{v})$ and $L \neq L_{final}$, for each $\hat{q} \in \hat{Q}_q(\mathbf{y})$, let $\hat{\mathcal{X}}_{\hat{q}} \subseteq \mathbb{R}^n$ be a set of plant states safe for some time bound $T_{\hat{q}}$ at $\hat{q}$. Let $T = \min_{\hat{q} \in \hat{Q}_q(\mathbf{y})} T_{\hat{q}}$ denote the minimum of the time bounds for each $\hat{q}$. The set*

$$\mathcal{X} = [\mathbf{y}]_{\approx_q} \cap \overline{Fail_q} \cap \bigcap_{\hat{q} \in \hat{Q}_q(\mathbf{y})} \hat{X}_{\hat{q}}$$

is safe for time bound T at q.

Proof. We prove this theorem by contradiction. Assume $\mathcal{X}$ is not safe for T at q. This means that there exists a plant state $\mathbf{z} \in \mathcal{X}$ and a trace $\sigma = s_0 s_1 \ldots s_K$, for some K, such that $s_0 = (q, \mathbf{z})$, $s_K \in Fail$, and $duration(\sigma) \leq T$. Since $\mathbf{z} \in \mathcal{X} \subseteq \overline{Fail_q}$, we know that $s_0 \notin Fail$ and therefore the trace must contain at least two states ($K \geq 1$). The first transition of σ must be a discrete transition because $L \neq L_{final}$ by hypothesis. Let $s_1 = (\hat{q}, \mathbf{z})$ and $\hat{\sigma} = s_1 \ldots s_K$. Since $\mathbf{z} \in \mathcal{X} \subseteq [\mathbf{y}]_{\approx_q}$, by hypothesis, we know that there exists a discrete transition $(q, \mathbf{y}) \longrightarrow (\hat{q}, \mathbf{y})$. Therefore, by hypothesis, $\hat{q} \in \hat{Q}_q(\mathbf{y})$. Since we assumed that $\mathbf{z} \in \mathcal{X}$ and, by hypothesis, $\mathcal{X} \subseteq \hat{\mathcal{X}}_{\hat{q}}$, we have that $\mathbf{z} \in \hat{\mathcal{X}}_{\hat{q}}$. But, by hypothesis, $\hat{\mathcal{X}}_{\hat{q}}$ is safe for time bound $T_{\hat{q}}$ at $\hat{q}$. This means that there does not exist any trace starting at $(\hat{q}, \mathbf{z})$ that reaches a state in $Fail$ and whose duration is less than or equal to $T_{\hat{q}}$. But $\hat{\sigma}$ is such a trace because $duration(\hat{\sigma}) \leq T_{\hat{q}}$. This is true since $duration(\hat{\sigma}) = duration(\sigma) \leq T_{\hat{q}}$, by assumption, and $T \leq T_{\hat{q}}$. This is a contradiction and therefore $\mathcal{X}$ must be safe for T at q. □

Figure 2-(a) shows an application of the theorem above. In this case, the state $s_1 = (q, \mathbf{y})$ has two successors, states s_2 and s_4. We assume that the sets N_2 and N_4 are safe for time bound t_s. The set X_1 in the figure denotes the equivalence class $[\mathbf{y}]_{\approx_q}$ corresponding to s_1. Then N_1 is safe for t_s, because N_1 does not intersect the fail states of q, every state of N_1 is program equivalent to $\mathbf{y}$, and N_1 is contained within both N_2 and N_4.

The conservative merging occurs when a trace reaches a state within a safe set of plant states. State s_7 in Figure 2-(b) is within N_2, which we assume to be safe for time bound t_s. The state s_6 has a single successor, namely s_7. The set X_6 in the figure denotes the equivalence class corresponding to state s_6. The set

N_6 does not intersect the fail region, N_6 is a subset of N_2, and N_6 is a subset of the equivalence class X_6. Therefore, by Theorem 3, we can deduce that N_6 is safe for t_s: any trace starting from a plant state within N_6 leads to a state within N_2.

3.1 Bounded-Time Safety Verification Algorithm

This section gives an algorithm to check bounded-time safety of an *SDCS*. This algorithm is based on the explicit-state Model Checking algorithm [4], but uses level sets of a bisimulation function and the notion of *program equivalence* to determine sets of plant states that are safe. The standard explicit-state Model Checking algorithm is a depth first search of the set of reachable states for each of the initial states. By using bisimulation functions and the notion of program equivalence, the algorithm presented here is able to determine, without looking at every trace, if a certain state encountered during the analysis is guaranteed not to lead to a fail state.

The procedures `main` and `explore` in Figure 3 implement the depth first search. For each initial state $(q, \mathbf{x})$, the procedure `explore` is invoked to perform a depth first search up the time bound T (lines 5-10). If the initial state is safe, a set of states that are safe for T at q is returned: this set is added to the set of initial states that are guaranteed to be safe (Safe_{Init} on line 8). Otherwise, if an error was detected, it is returned immediately (line 10). After analyzing each initial state, the set of safe initial states is returned on line 11. The procedure `explore` takes as arguments a state $(q, \mathbf{x})$, a time bound τ, and a trace σ which leads to $(q, \mathbf{x})$. The time bound τ represents the amount of time remaining from the given state, that is $\tau = T - \mathit{duration}(\sigma)$. It performs the actual depth first search starting from the given state up to the time bound. The trace σ is used to generate a counterexample if a fail state is reached (line 13). The current state is compared with the sets of safe states that have been determined so far (lines 14-15). If there exists a set of plant states $\hat{\mathcal{X}}$ that is safe for the current supervisor state q and a longer time bound $\hat{\tau} \geq \tau$, the search of this branch can terminate and the set of plant states $\hat{\mathcal{X}}$ is returned to the caller as safe.

Two ways of computing the successor states are possible. If the current control state is equal to L_{final}, then a plant transition is performed by calling the function `plant_transition` (line 17). Otherwise, the transitions of the supervisor are explored by calling the function `supervisor_transitions` (line 19). In either case, if the result is that the current state is safe, the set $\mathcal{X}$ of plant states that are computed to be safe for state q and time bound τ is added to the list of safe sets (line 21).

The result of a plant transition is computed by the function `plant_transition` in Figure 3. Line 25 is executed if the time bound has been reached, i.e., there is not enough time left to complete an additional plant transition. The set of plant states that are safe for time bound τ at q is simply the set of plant states that are not fail states at supervisor state q, since $\tau < t_s$ (line 24). Otherwise, the successor state $(\hat{q}, \hat{\mathbf{x}})$ of the current state $(q, \mathbf{x})$ is computed using numerical simulation (line 26) and by setting the current control location to $L_{initial}$ (line

1: **global** SDCS, *Fail*, T;
2: **global** safe_sets $\leftarrow \emptyset$; // *Sets of safe plant states, initially empty.*

3: **main**: // *Check bounded-time safety of SDCS*
4: Safe$_{Init} \leftarrow \emptyset$ // *Set of safe initial states.*
5: **foreach** ((q, **x**) $\in$ Init) // *Depth-first search for each initial state.*
6: result $\leftarrow$ explore(q, **x**, T, [(q, **x**)]);
7: **if**(result = (*SAFE*, $\mathcal{X}$))
8: Safe$_{Init}$ $\leftarrow$ Safe$_{Init}$ $\cup$ $\{(q, \mathbf{x}) \mid \mathbf{x} \in \mathcal{X}\}$ // *Add to safe initial states.*
9: **else**
10: **return** result; // *An error was detected.*
11: return (*SAFE*, Safe$_{Init}$); // *Return the set of safe initial states.*

12: **function** explore(q, **x**, τ, σ) // *Depth-first search from* (q, **x**) *up to time* τ.
13: **if** ((q, **x**) $\in$ *Fail*) **return** (*UNSAFE*, σ); // *Check for fail states.*
14: **if** ($\exists$ ($\hat{q}$, $\hat{\mathcal{X}}$, $\hat{\tau}$) $\in$ safe_sets: $q = \hat{q} \wedge \mathbf{x} \in \hat{\mathcal{X}} \wedge \tau \leq \hat{\tau}$)
15: **return** (*SAFE*, $\hat{\mathcal{X}}$); // *Merge traces if within a safe set.*
16: **if** (q.L = L_{final})
17: result = plant_transition(q, **x**, τ, σ); // *Plant transition*
18: **else**
19: result = supervisor_transitions(q, **x**, τ, σ); // *Supervisor transitions*
20: **if** (result = (*SAFE*, $\mathcal{X}$))
21: safe_sets $\leftarrow$ safe_sets $\cup$ $\{(q, \mathcal{X}, \tau)\}$; // *Plant states safe for* τ *at* q.
22: **return** result;

23: **function** plant_transition(q, **x**, τ, σ)
24: **if** ($\tau < \mathrm{t}_s$) // *Stop if time bound is less than sampling time*
25: **return** (*SAFE*, $\{\mathbf{x} \mid (q, \mathbf{x}) \notin$ *Fail*$\}$);
26: $\hat{\mathbf{x}} \leftarrow \mathrm{sim}(\mathbf{x}, \mathrm{f}_{q.\mathbf{v}})$; // *Numerical simulation*
27: $\hat{q} \leftarrow (L_{initial}, q.\mathbf{v})$;
28: result = explore($\hat{q}$, $\hat{\mathbf{x}}$, τ - t_s, $\sigma \cdot (\hat{q}, \hat{\mathbf{x}})$);
29: **if** (result = (*SAFE*, $\hat{\mathcal{X}}$))
30: $r_{max} \leftarrow \max\left\{r \mid \mathcal{N}_{\varphi_{q.\mathbf{v}}}(\hat{\mathbf{x}}, r) \subseteq \hat{\mathcal{X}}\right\}$; // *Safe set of plant states*
31: **return** (*SAFE*, $\mathcal{N}_{\varphi_{q.\mathbf{v}}}(\mathbf{x}, r_{max})$);
32: **else**
33: **return** result;

34: **function** supervisor_transitions(q, **x**, τ, σ)
35: $\hat{Q} \leftarrow \{\hat{q} \mid \exists i$: $(q, \hat{q}) \in \delta_i(\mathbf{x})\}$; // *Explore each successor*
36: $\mathcal{X} \leftarrow [\mathbf{x}]_{\approx_q} \cap \overline{Fail}$;
37: **foreach** ($\hat{q} \in \hat{Q}$)
38: result $\leftarrow$ explore($\hat{q}$, **x**, τ, $\sigma \cdot (\hat{q}, \mathbf{x})$);
39: **if** (result = (*SAFE*, $\hat{\mathcal{X}}$))
40: $\mathcal{X} = \mathcal{X} \cap \hat{\mathcal{X}}$;
41: **else**
42: **return** result;
43: return (*SAFE*, $\mathcal{X}$);

Fig. 3. The conservative merging verification algorithm.

27). The search continues from the new state by calling `explore`. The recursive call uses a smaller time bound and adds one state to the trace being constructed (line 28). If the result at line 28 is that state $(\hat{q}, \hat{\mathbf{x}})$ is safe, the set of states $\hat{\mathcal{X}}$ that are safe for time bound $\tau - t_s$ at $\hat{q}$ is used to determine the maximum size of a sublevel set of the bisimulation function centered around $\mathbf{x}$ that is safe for time bound τ at q by solving the optimization problem:

$$r_{max} = \max \left\{ r \in \mathbb{R} \,|\, \mathcal{N}_{\varphi_{\mathbf{v}}}(\hat{\mathbf{x}}, r) \subseteq \hat{\mathcal{X}} \right\},$$

where $\mathbf{v}$ is the current value of the supervisor variables and $\varphi_{\mathbf{v}}$ is the bisimulation function for $\dot{\mathbf{x}} = f_{\mathbf{v}}(\mathbf{x})$ (line 30). The set $\mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{x}, r_{max})$ is returned to the caller since it is safe for time bound τ at q.

The function `supervisor_transitions` in Figure 3 computes and explores the successors of a state $(q, \mathbf{x})$ that originate from transitions of the supervisor. The set of successors $\hat{Q}$ is generated by using the transition relations $\delta_1, \ldots, \delta_p$ of the tasks that make up the supervisor (line 35). Each successor $\hat{q}$ is visited by calling the function `explore` over $(\hat{q}, \mathbf{x})$ with the same time bound τ (since supervisor transitions are instantaneous) and with a trace that adds the new state $(\hat{q}, \mathbf{x})$ to σ (line 38). If the state $(\hat{q}, \mathbf{x})$ is safe, a set of safe plant states $\hat{\mathcal{X}}$ is returned by the recursive call. The set of safe plant states that is returned to the caller by this call (line 43), computed by lines 36 and 40, is

$$X = [\mathbf{x}]_{\approx_q} \cap \overline{Fail_q} \cap \bigcap_{\hat{q} \in \hat{Q}} \hat{\mathcal{X}}_{\hat{q}}.$$

This concludes the description of the algorithm. The following theorems establish correctness and termination of the procedure.

Theorem 4 (Correctness). *Consider an SDCS, a set of fail states Fail, and a time bound T. If the algorithm of Figure 3 returns (SAFE,* Safe_{Init}*) then the SDCS is safe for time bound T, Init $\subseteq$* Safe_{Init}*, and all states in* Safe_{Init} *are safe for time bound T. If the algorithm returns (UNSAFE, σ) then SDCS is not safe for time bound T and σ is a trace of duration less than T that ends at a state in Fail.*

Proof. (Sketch) We can show, by induction on the time bound τ that, given a state $(q, \mathbf{x})$ and a path σ to $(q, \mathbf{x})$, if `explore` returns (SAFE, $\mathcal{X}$), then $\mathbf{x} \in \mathcal{X}$ and $\mathcal{X}$ is safe for τ at q.

Three cases are possible. The base case is when $\tau < t_s$ and $q = (L_{final}, \mathbf{v})$: in this case, the algorithm computes a set of states that is safe for τ at q. Otherwise, if $q = (L_{final}, \mathbf{v})$ and $\tau \geq t_s$, a continuous transition is possible: in this case, by Theorem 2, the computed set is safe for τ at q. If the location of q is not equal to L_{final}, the set of supervisor transitions is computed and, according to Theorem 3, the constructed set is safe for τ at q.

From this, we can deduce that Safe_{Init} contains every initial state of *SDCS* and that each state in Safe_{Init} is safe for time bound T.

If the algorithm returns (UNSAFE, σ), the trace σ is a valid trace from an initial state to a fail state and its duration is less than T by construction. This trace proves that *SDCS* is unsafe. □

Theorem 5 (Termination). *Given an SDCS* $= \langle \{\mathcal{T}_1, \ldots, \mathcal{T}_p\}, V, f_{\mathbf{v}}, t_s, Init \rangle$ *such that Init is finite, the algorithm of Figure 3 always terminates.*

Proof. (Sketch) Since we assumed that all sequences of supervisor transitions are finite, then any trace of duration less than or equal to T must have finite length. Since the plant transitions are deterministic and there exists a finite set of supervisor states, since *Loc* and V are finite, the recursion tree obtained during execution of the algorithm has a finite maximum degree. Since the depth of the tree is finite and the degree of each node is finite, there exists a finite number of nodes, and the algorithm will terminate after exploring each node. □

3.2 Ellipsoidal Sets for Affine Dynamics

In this subsection, we discuss properties related to our technique for the case of stable affine plant dynamics.

Bisimulation Functions. For the special case of affine plant dynamics, that is $f_{\mathbf{v}}(\mathbf{x}) = \mathbf{A_v x} + \mathbf{B_v}$, a bisimulation function is given by

$$\varphi_{\mathbf{v}}(\mathbf{y}, \mathbf{z}) = (\mathbf{z} - \mathbf{y})^T \mathbf{P_v} (\mathbf{z} - \mathbf{y}),$$

where $\mathbf{P_v}$ satisfies the Lyapunov inequality ${\mathbf{A_v}}^T \mathbf{P_v} + \mathbf{P_v A_v} \leq \mathbf{0}$. The level sets are given by $\mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{x}, r) = \{\mathbf{z} \in \mathbb{R}^n \mid (\mathbf{z}-\mathbf{x})^T \mathbf{P_v} (\mathbf{z}-\mathbf{x}) \leq r\}$, which are ellipsoidal.

Maximum Ellipsoid Within an Ellipsoid. In the case of affine dynamics, an operation required by the procedure given in Section 3.1 is the computation of the maximum sized ellipsoid contained in a second ellipsoid. Given a set $\mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{z}, r_{\mathbf{z}})$ and a point $\mathbf{y} \in \mathbb{R}^n$, we want to find the maximum $r_{\mathbf{y}}$ such that $\mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{y}, r_{\mathbf{y}}) \subseteq \mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{z}, r_{\mathbf{z}})$. It is shown in [2] that this is equivalent to the following:

$$\begin{aligned} &\max_{\lambda, c} \; c \\ &\text{s.t.} \; \begin{bmatrix} -r_{\mathbf{z}} \mathbf{Q_v} & (\mathbf{z}-\mathbf{y}) & c\sqrt{\mathbf{Q_v}} \\ (\mathbf{z}-\mathbf{y})^T & \lambda - 1 & 0 \\ c\sqrt{\mathbf{Q_v}} & 0 & -\lambda \mathbf{I} \end{bmatrix} \leq \mathbf{0}, \lambda \geq 0, c \geq 0, \end{aligned}$$

where $c = \sqrt{r_{\mathbf{y}}}$, $\mathbf{Q_v} = {\mathbf{P_v}}^{-1}$, $\mathbf{I}$ is the identity matrix, and $\sqrt{\mathbf{Q_v}}$ is the matrix that satisfies $\mathbf{Q_v} = \sqrt{\mathbf{Q_v}}\sqrt{\mathbf{Q_v}}$, which exists since $\mathbf{Q_v}$ is positive semidefinite. This can be formulated as a convex problem with LMI constraints. Numerical tools exists for solving such problems in polynomial time.

Maximum Ellipsoid Within a Set of Linear Constraints. The other operation required by the procedure given in Section 3.1 in the case of affine dynamics is the computation of an ellipsoid of maximum size that satisfies a conjunction

of linear constraints. We want to maximize r subject to constraints of the form $\bigwedge_{i=1}^{i_{max}} \mathbf{c}_i^T \mathbf{y} \leq b_i$ for all $\mathbf{y} \in \mathcal{N}_{\varphi_\mathbf{v}}(\mathbf{x}, r) = \{\mathbf{z} \in \mathbb{R}^n \mid (\mathbf{z}-\mathbf{x})^T \mathbf{P}_\mathbf{v} (\mathbf{z}-\mathbf{x}) \leq r\}$, where $b_i \in \mathbb{R}$, $\mathbf{c}_i \in \mathbb{R}^n$ for each i. Let $\mathbf{Q}_\mathbf{v} = {\mathbf{P}_\mathbf{v}}^{-1}$. The maximum r that satisfies the linear constraints is then given by [12]

$$r^* = \min_{i \in \{1,\ldots,i_{max}\}} \frac{(b_i - \mathbf{c}_i^T \mathbf{x})^2}{\mathbf{c}_i^T \mathbf{Q}_\mathbf{v} \mathbf{c}_i}.$$

4 Experimental Results

The technique presented in the previous section was implemented using an existing explicit-state source-code Model Checker. The tool we chose is Java PathFinder [16]. While the main purpose of the tool is to verify Java programs, it handles the subset of C that is common to the two languages. This prototype implementation handles systems where the plant dynamics are affine. We use the LMI tool CVX with the semidefinite program solver SDPT3 [7, 15] to solve the optimization problems that arise during the verification.

Java PathFinder was extended as follows. The state of a system was enhanced to include the plant state $\mathbf{x}$, represented by a set of floating-point variables. Our extension stores sets of plant states that are safe with respect to a given supervisor state and time bound. Safe sets are represented as ellipsoidal sets, and program equivalence classes and system requirement are represented as sets of linear constraints. Ellipsoidal sets are represented by their size parameter r and their center, while the shape and orientation are determined by the bisimulation function given for each set of plant dynamics. The set of constraints used to express the set of fail states as well as the program equivalence classes are given as annotations. Moreover, since the plant dynamics are affine, it is possible to convert the continuous-time dynamics into discrete-time difference equations over the fixed sampling period t_s.

We applied our technique to an example based on the Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC), a quadrotor unmanned aerial vehicle (UAV) under development at Stanford University [9]. The vehicle, shown in Figure 4, is composed of a computer controller and power supply at its center, which is attached to a frame on which four rotors are mounted. A computer controller sends thrust commands to the four rotors. The supervisor makes its decisions based on measurements of the state of the vehicle. We consider a model of the STARMAC system containing six plant state variables: the horizontal position and velocity (x and $\dot{x}$), the vertical position and velocity (z and $\dot{z}$), and the rotation about the y-axis and the corresponding rotational velocity (θ and $\dot{\theta}$). The y position and rotation around the x-axis and z-axis are not included in this model. Motors 1 and 3 provide lift and torque around the y-axis, while motors 2 and 4 only provide lift. The forces applied by motors 2 and 4 lie on the y-axis and are not shown in Figure 4. Equivalent force is applied by motors 2 and 4 at all times.

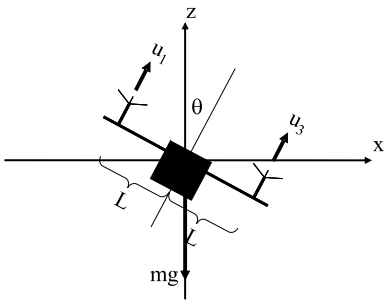

Fig. 4. An illustration of the dynamics captured by the enhanced STARMAC model.

The equations of motion are

$$\ddot{x} = -\frac{b}{m}\dot{x} + \frac{1}{m}(u_1 + u_2 + u_3 + u_4)\sin(\theta)$$

$$\ddot{z} = -\frac{b}{m}\dot{z} + \frac{1}{m}(u_1 + u_2 + u_3 + u_4)\cos(\theta) - g$$

$$\ddot{\theta} = \frac{L}{I_y}(u_1 - u_3) - \frac{c}{I_y}\dot{\theta},$$

where $L = 0.236\,\mathrm{m}$ is the distance from the center of the UAV to each motor, $b = 0.3\,\mathrm{N}\cdot\mathrm{sec/m}$ is the viscous damping due to translation, $m = 0.518\,\mathrm{kg}$ is the mass of the UAV, $I_y = 0.048\,\mathrm{kg}\cdot\mathrm{m}^2$ is the moment of inertia of the UAV about the y-axis, $c = 0.03\,\mathrm{N}\cdot\mathrm{m}\cdot\mathrm{sec/rad}$ is the rotational viscous damping, u_i are the forces produced from each of the four motors, and g is the acceleration due to gravity in meters per second squared. We linearized the equations and designed a linear quadratic regulator (LQR) to drive the system to a given set point. The LQR controller is modeled as part of the plant. The system we obtained is of the form $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{x}^*$, where $\mathbf{x}^*$ is the set point we want to reach and

$$\mathbf{A} = -\mathbf{B} = \begin{bmatrix} -0.6 & 0.0 & 0.0 & 0.0 & 0.0 & 9.8 \\ 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -1.1 & -0.4 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ -35.4 & -22.1 & 0.0 & 0.0 & -70.2 & -2221.7 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix}.$$

The supervisory controller for this system is implemented by two concurrent tasks: one task determines the target position based on a given list of waypoints; the other sends position commands to the plant. Due to the interleaving of the two tasks, the plant might receive the updated target position with a sampling period delay, and the system will follow slightly different traces every time a new waypoint is generated.

We performed the analysis both with and without state merging. The results, presented in Figure 5 show a significant reduction in number of visited states and memory usage. The space overhead due to the ellipsoidal sets that need to be associated with each visited state was limited and it was offset by the reduction in memory consumption due to the drastic reduction in number of visited states. Such a reduction was obtained with just a handful of conservative state merges: even a single merge can lead to a large reduction because every state reachable from the merged state no longer needs to be visited. The approach as implemented showed a significant overhead in terms of running time, however, which could be reduced by further optimizing the operations involving storing and lookup of ellipsoids.

	Model-Checking-Guided Simulation without Merging	Model Checking with Safe Sets and Merging
Visited states	43,134	25,493
Running time	17 sec	107 sec
Memory usage	90.2MB	77.0MB

Fig. 5. Visited states, running time, and memory usage for the time bound $T = 90\,\text{sec}$ with and without merging of safe states. The number of state merges was 282.

5 Conclusions

This paper presents a formal verification technique for embedded control systems based on the combination of software Model Checking and numerical simulation of a continuous dynamic plant. The technique can provide a guarantee that a continuous dynamic plant controlled by a supervisor implemented in software satisfies safety requirements. The power of a software Model Checker lies in its ability to efficiently analyze systems with complex behaviors that are due to program concurrency. This feature is exploited by our technique by using the Model Checker to explore the nondeterministic behaviors of a controller that are due to, for example, interleaving of tasks. The technique uses a notion of program equivalence for the behaviors of the controller and bisimulation functions for the behaviors of the plant to merge states. This leads to a reduction in the number of states that need to be visited in order to prove safety.

The algorithm presented in this work can be applied to system with nonlinear plant dynamics; however, the optimization problems involved can only be solved efficiently under certain assumptions. The work by Parrilo et al. on identifying Lyapunov functions for a class of nonlinear system is related to the work presented here [14]. Their technique uses semidefinite programming to compute sum of squares functions that satisfy Lyapunov equations. Similar techniques can be applied to determine bisimulation functions, since they satisfy similar

constraints. The level sets of these functions can provide safety guarantees for trajectories that are in proximity to each other using the technique described in this paper.

References

1. Erin Aylward, Pablo A. Parrilo, and Jean-Jacques E. Slotine. Algorithmic Search for Contraction Metrics via SOS Programming. In *Proc. of the 2006 American Control Conference*, 2006.
2. Stephen Boyd, Laurent E. Ghaoui, Eric Feron, and Venkataramanan Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*, volume 15 of *SIAM Studies in Applied Mathematics*. SIAM, 1994.
3. 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.
4. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.
5. Alexandre Donzé and Oded Maler. Systematic Simulation using Sensitivity Analysis. In *Proc. of the 10th International Workshop on Hybrid Systems: Computation and Control*, 2007.
6. Antoine Girard and George J. Pappas. Approximation Metrics for Discrete and Continuous Systems. Technical Report MS-CIS-05-10, University of Pennsylvania, 2005.
7. Michael Grant, Stephen Boyd, and Yinyu Ye. *CVX User's Guide*. 2007.
8. Thomas A. Henzinger. The Theory of Hybrid Automata. In *Proc. of the 11th Annual IEEE Symposium on Logic in Computer Science*, 1996.
9. 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.
10. A. Agung Julius, Georgious E. Fainekos, Madhukar Anand, Insup Lee, and George J. Pappas. Robust Test Generation and Coverage for Hybrid Systems. In *Proc. of the 10th International Workshop on Hybrid Systems: Computation and Control*, 2007.
11. James Kapinski, Bruce H. Krogh, Oded Maler, and Olaf Stursberg˙ On Systematic Simulation of Open Continuous Systems. In *Proc. of the 6th International Workshop on Hybrid Systems: Computation and Control*, 2003.
12. Alexander B. Kurzhanski and Istvan Vályi. *Ellipsoidal Calculus for Estimation and Control*. Birkhäuser, Boston, 1997.
13. Flavio Lerda, James Kapinski, Hitashyam Maka, Edmund M. Clarke, and Bruce H. Krogh. Model Checking In-The-Loop. In *Submitted to the 27th American Control Conference*, 2007.
14. Pablo A. Parrilo. *Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization*. PhD thesis, California Institute of Technology, 2000.
15. Kim-Chuan Toh, Michael J. Todd, and Reha H. Tütüncü. *SDPT3 4.0*. MIT Press, 2006.
16. Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.