# Verification of Embedded Control Software

Flavio Lerda

December 12, 2007

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements*
*for the degree of Doctor of Philosophy.*

**Thesis Committee:**
Edmund M. Clarke - chair
Stephen Brookes
Bruce Krogh
Rajeev Alur (University of Pennsylvania)



The author was supported by General Motors and the Carnegie Mellon University-General Motors Collaborative Research Laboratory under grant no. GM9100096UMA.

This research was sponsored by the Office of Naval Research (ONR), the Naval Research Laboratory (NRL), the Army Research Office (ARO), the Air Force Research Office (AFRO), and the National Science Foundation (NSF).

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring institutions, the U.S. Government or any other entity.

**Keywords:** Formal methods, model checking, numerical simulation, hybrid systems, control software.

# Abstract

Embedded control software is ubiquitous nowadays. It is a significant component of, for instance, home appliances, cars, and medical devices. As the uses of software increase in our daily life, the importance of its correctness increases as well. At the same time, expectations for embedded control software are usually higher than for desktop applications, making correctness a crucial problem in this domain.

One approach to improve the reliability of embedded control software is by means of validation and verification techniques, which analyze a system and try to determine if the given requirements are satisfied. Techniques like numerical simulation check that the system behaves correctly by exploring a number of its possible behaviors. Techniques like model checking, on the other hand, formally establish the validity of properties of the system.

The focus of this thesis is the combination of software model checking with numerical simulation for validation and verification of embedded control software. A characteristic of embedded control software is the interaction with an environment that is continuous in nature. This makes model checking

more difficult as the system has an infinite number of states.

This thesis aims at developing different techniques that can be used to analyze control systems. These techniques range from an optimized version of numerical simulation to a conservative approach for the verification of control system's safety. The techniques not only differ in the type of results they can provide (correctness versus bug finding), but also in the associated complexity. I see these approaches as a first step toward developing a spectrum of techniques with different costs that can be applied to analyze control systems.

In this proposal, I first describe an approach called *systematic simulation* that employs a model checker to explore the behaviors of a software implementation while using MATLAB/Simulink to model and simulate the continuous environment. This approach is used to perform bounded-time verification of an embedded control system with a finite set of initial states whose controller is implemented in software. Next, I present an approach based on a notion of *approximate equivalence* between states of the systems that can be used to test an embedded control system by determining during the analysis which traces to explore. Last, I present a conservative extension of the latter approach that is able to formally verify bounded-time safety of embedded control software. This approach performs a *conservative merging* of traces that are similar, based on a conservative notion of equivalence between states. So far I have applied these techniques to a few control system examples including the design of an unmanned aerial vehicle (UAV) based

on the Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC).

# Table of Contents

# Chapter 1

# Overview

Model-based design of embedded control systems is becoming standard practice. The goal of model-based design is to reduce development time and cost by evaluating controllers using computer-based models before building the actual system. 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. 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, called *the plant*. In this thesis, I propose to investigate the use of formal verification techniques to catch design errors and verify correctness of a design. The approaches presented here explore

the behaviors of an embedded control system using techniques that combine software model checking with numerical simulation.

*Numerical 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 [24, 1]. Simulation requires the developer to specify a set of cases that need to be checked. The effectiveness of simulation, therefore, depends on the ability of the developer to identify a representative set of cases. The results of simulation are limited to the cases that have been explicitly checked, however. While numerical simulation may be able show the presence of errors, it is unable to prove the absence thereof.

*Model checking* is an automated technique for the formal verification of temporal properties [8, 9]. One of the main advantages of model checking compared to other techniques, such as numerical simulation, is the ability to explore behaviors exhaustively and therefore prove the correctness of a system. In recent years, there has been considerable interest in model checkers for software [2, 14, 5, 27]. These techniques aim at checking correctness of software systems expressed in modern programming languages. These approaches, however, cannot be applied directly to embedded control software, which are difficult to analyze using model checking due to the controller's interaction with a continuous dynamic plant.

This thesis aims at developing approaches that narrow the gap between simulation and model checking of embedded control software. Using numer-

ical simulation, these approaches capture the dynamics of the continuous dynamic plant accurately. By employing a model checker, they are able to verify properties that are difficult to verify using standard simulation. In this thesis, I propose three different approaches. The first one, called *systematic simulation*, is an extension of standard numerical simulation, which uses a model checker to explore a set of possible behaviors systematically. The second approach, called *path pruning*, extends systematic simulation by using a notion of *approximate equivalence* to reduce the number of paths that need to be explored during the analysis. The latter approach is not conservative, however, as it uses an approximate notion of equivalence based on a measure of the proximity of states. The third part of my thesis proposes an approach, called *conservative merging*, that is able to reduce the number of paths that are explored during the verification while guaranteeing that, if no error is detected, the system is safe.

## 1.1 Systematic Simulation

Numerical simulation is a validation technique that generates a trace of a system. For a dynamical system specified as a set of differential equations, numerical methods are used to perform the simulation. A simulation trace corresponds to one possible evolution of the dynamical system: all inputs must be fixed, and therefore the simulation is deterministic.

In order to check that the system behaves correctly for all values of the

inputs and deal with non-deterministic behavior, I propose to use a model checker to guide a numerical simulation engine to produce all possible traces of a given system. If the set of initial states and inputs is finite, this allows us to prove that the system is safe for all possible values of the inputs and all possible non-deterministic behaviors. By using a model checker, the technique is more efficient than using standard simulation. While simulation explores traces independently of each other, the traces explored by a model checker form a state-transition graph called the state-space graph. Model checkers explore each transition of the state-space graph only once, reducing the amount of computation required when compared to standard simulation.

This thesis considers a particular type of control systems where the controller is implemented in software as a set of concurrent and/or distributed tasks. We have implemented a tool based on an explicit-state model checker for software that uses MATLAB/Simulink as the numerical simulation engine [21]. We applied this approach to a MATLAB/Simulink model of an unmanned aerial vehicle and was able to detect an error in the controller. This work has been done in collaboration with Dr. James P. Kapinski, Hitashyam Maka, my advisor Prof. Edmund M. Clarke, and Prof. Bruce H. Krogh.

## 1.2 Path Pruning

The systematic simulation approach exhaustively explores all possible behaviors of a system. This may require 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. The *path pruning* approach presented in this thesis tackles this problem by pruning parts of the state-space graph generated by the model checker.

Explicit-state model checkers compute the set of reachable states iteratively by constructing the state-space graph using the transition relation of the system. When the model checker encounters a state that is equal to a previously visited state, the successors of the current state are not explored. Doing so would only lead to states that have already been encountered. The *path pruning* approach replaces state equality used in standard model checkers with a notion of 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 search for counterexamples efficiently, it does not explore all possible system behaviors. As such it can show that a system is unsafe, but it cannot prove that a system is safe.

So far, we have developed different notions of approximate equivalence and applied them to a robotics control system example [25]. The work has

been developed in collaboration with Sebastian Scherer and my advisor Prof. Edmund M. Clarke.

## 1.3 Conservative Merging

The path pruning approach presented above is based on a notion of *approximate equivalence* that is used to prune the state space. The approach is not conservative, however; it can be used to find counterexamples, but it is unable to prove safety. In this thesis I also propose an approach called *conservative merging* that is able to prove bounded-time safety of a system while performing path pruning, 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 *conservative merging* when two states are in proximity to each other if the pruned parts of the state space are guaranteed to be safe. In this thesis, I show how to determine safe sets of plant states around each state of a trace. These sets correspond to a set of traces that are in proximity of the explored trace and are guaranteed to be safe. When a state that is within a safe set is reached, the current state can be merged conservatively with a previously visited state 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 thesis uses bisimulation functions [11] 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 [11, 16]. In this work, we use bisimulation functions to approximate conservatively the plant transitions while using a notion of *program equivalence* to approximate conservatively the behavior of the controller.

We have implemented this technique for the special case of affine plant dynamics, for which an efficient algorithm to compute bisimulation functions exists, and used this technique to prove the safety of a model of an unmanned aerial vehicle [20]. This work has been done in collaboration with Dr. James P. Kapinski, my advisor Prof. Edmund M. Clarke, and Prof. Bruce H. Krogh.

## 1.4 Related Work

This thesis presents three different techniques that can be used to analyze control systems. They range from an optimized version of numerical simulation (*systematic simulation*) to a conservative verification technique (*conservative merging*). The techniques not only differ in the type of results they can provide (bug finding versus correctness), but also in the associated complexity. These three technique are a first step toward developing a range of techniques that can be applied to analyze control systems.

Various methods have been developed to verify hybrid automata, which

can model embedded control systems [13, 18, 3, 7]. These techniques are computationally expensive, however, and are able to analyze only systems of low complexity. A central problem in verifying safety properties of control systems is reachable set estimation, i.e., the problem of determining the set of states that are reachable from a given set of initial states.

Other verification techniques for control systems that are based on numerical simulation exist. For example, Kapinski et al. [17] have investigated the use of ellipsoidal sets to overapproximate the set of reachable states of a dynamical system. Donzé and Maler [10] have showed how to use sentitivity analysis to perform reachability analysis. Both approaches perform a search forward in time while computing an overapproximation of the set of reachabile states. The work in this thesis instead uses the safety requirements to construct sets of states that are guaranteed to be safe, proceeding backward in time. The work by Julius et al. also provides a means for determining maximum safety bounds for simulation traces [16], but the technique presented here goes further by handling more complex, nondeterministic control systems. This is especially important when modeling a controller that is implemented in software where external inputs and task interleaving, due for instance to a distributed implementation, can lead to nondeterministic behavior. The work presented here deals efficiently with the large number of reachable paths that occur due to nondeterministic behaviors in the controller.

The rest of the proposal is organized as follows. Chapter 2 describes the

model of control systems considered in this proposal. Chapter 3 describes a technique called *systematic simulation* that combines a software model checker and numerical simulation to verify bounded-time safety of embedded control software with a finite set of initial states. Since the number of traces of a system may be quite large, Chapter 4 introduces an approach, based on a notion of *approximate equivalence* between states of the system, that is able to prune some traces that are similar to previously explored traces. This approach reduces the amount of work needed to analyze a system, but, like simulation, it only looks at a subset of the possible behaviors, and, therefore, it is unable to prove correctness. While this approach is useful to analyze a system and discover errors, Chapter 5 proposes a different approach that is able to *merge states conservatively* and therefore guarantees the correctness of the system. Lastly, Chapter 6 contains a summary of the work that still needs to be done for this thesis and a timeline for its completion.

# Chapter 2

# System Model

## 2.1 Control Systems

A control system is a system composed of a continuous-time environment that is being controlled, called the *plant*, and a *controller*. Usually, the controller is able to sense the outputs of the plant (called the *sensor values*) and decides the commands to send to the plant (called the *actuator values*). This type of control system is usually referred to as a *closed-loop control system*. In this work, I will consider closed-loop control systems where the controller is implemented in software. Since the software executes only at discrete times, I will look at a particular class of control systems known as *sampled-data control systems*. A *sampled-data controller* is able to observe the state of the plant only at discrete time instants, called *sample times*. I assume here, without loss of generality, that the sample times occur at multiples of a fixed

*sampling period*, $t_s > 0$.

Let us consider a system made of two components: a continuous-time plant and a discrete controller. The plant has a set of $n$ real-valued state variables (*the plant state*, denoted by $\mathbf{x} \in \mathbb{R}^n$), a set of $s$ real-valued inputs (*the actuator values*, denoted by $\mathbf{u} \in \mathbb{R}^s$), and a set of $r$ real-valued outputs (*the sensor values*, denoted by $\mathbf{y} \in \mathbb{R}^r$). The plant evolves according to the differential equation $\dot{\mathbf{x}} = f_{\mathbf{u}}(\mathbf{x})$, where the dynamics of the plant depend on the actuator values $\mathbf{u}$. The sensor values $\mathbf{y}$ are a function of the plant state, i.e., $\mathbf{y} = g(\mathbf{x})$ for a fixed function $g$, independent of $\mathbf{u}$. The controller is represented as a finite-state machine with an initial location $L_{initial}$, a final location $L_{final}$, and a set of $m$ variables that assume values from a finite domain $V$ (*the controller variables*, denoted by $\mathbf{v} \in V^m$). At each sample time, the controller starts from the initial location $L_{initial}$ and continues executing until it reaches the final location $L_{final}$. The transitions of the finite-state machine may depend on and update the variables $\mathbf{v}$. Figure 2.1 shows the structure of this type of systems.

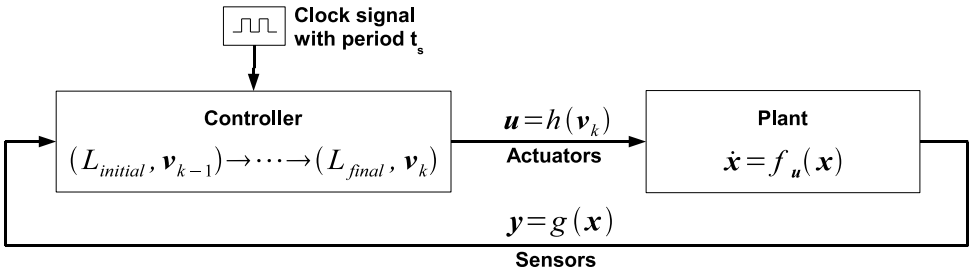



Figure 2.1: The architecture of a sampled-data system.

## 2.2 Sampled-Data Control System

This section presents a formal model of sampled-data control systems where the controller is implemented as a set of concurrent tasks. The model presented here is general enough that it can be used both for tasks that are executing on the same processor sharing variables and tasks that are distributed across multiple processors. In the following, I assume that the execution time of the transitions of the different tasks, the clock skew between different processors, and the communication time are small compared to the sampling period. This assumption is reasonable for most control systems implemented in software where the sampling time is very large compared to the clock cycle of modern processors.

**Definition 2.1 (Controller Task)** *Given a set of controller variables* $V^m$ *and a set of sensors values* $\mathbb{R}^r$, *a* controller task *is a tuple*

$$Task_i = \langle Loc_i, l_{i,initial}, l_{i,final}, \delta_i \rangle$$

*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* $Task_i$*; and*
- $\delta_i : \mathbb{R}^r \rightarrow 2^{Loc_i \times V^m \times Loc_i \times V^m}$ *is the* transition relation *of* $Task_i$*. 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 $l_{i,final}$. Notice that the transition relation $\delta_i$ depends on the current sensor values $\mathbf{y}$. Given $l_i, \hat{l}_i \in Loc_i$, $\mathbf{v}, \hat{\mathbf{v}} \in V^m$, and $\mathbf{y} \in \mathbb{R}^r$, there exists a transition from $(l_i, \mathbf{v})$ to $(\hat{l}_i, \hat{\mathbf{v}})$ when the sensor values are equal to $\mathbf{y}$ if and only if $(l_i, \mathbf{v}, \hat{l}_i, \hat{\mathbf{v}}) \in \delta_i(\mathbf{y})$.

**Definition 2.2 (Sampled-Data Control System)** *A sampled-data control system is a tuple*

$$SDCS = \langle \{ Task_1, \ldots, Task_p \}, V, h, f_{\mathbf{u}}, g, t_s, Init \rangle$$

*where:*

- $\{ Task_1, \ldots, Task_p \}$ *is a finite set of controller tasks;*
- $V$ *is a finite domain for the controller variables;*
- $h : V^m \rightarrow \mathbb{R}^s$ *is the actuator function that maps the values of the controller variables* $\mathbf{v}$ *into the actuator values* $\mathbf{u}$*;*
- *For each actuator value* $\mathbf{u} \in \mathbb{R}^s$, $f_{\mathbf{u}} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ *is a Lipschitz continuous function that describes the flow of the plant;*
- $g : \mathbb{R}^n \rightarrow \mathbb{R}^r$ *is the sensor function that maps a plant state* $\mathbf{x}$ *into the corresponding sensor values* $\mathbf{y}$*;*
- $t_s > 0$ *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.*

**Definition 2.3 (State)** *A state of an SDCS is a tuple* $(q, \mathbf{x})$ *where* $q = (L, \mathbf{v})$ *is the controller state,* $L \in Loc_1 \times \ldots \times Loc_p$ *specifies the control locations of each task,* $\mathbf{v} \in V^m$ *is the value of the controller variables, and* $\mathbf{x} \in \mathbb{R}^n$ *is the plant state.*

Given a controller state $q$, let us denote by $L(q)$ the control locations corresponding to $q$ and by $\mathbf{v}(q)$ the value of the controller variables corresponding to $q$. Let $L_{initial} = (l_{1,initial}, \ldots, l_{p,initial})$ denote the initial control location of the controller, and $L_{final} = (l_{1,final}, \ldots, l_{p,final})$ denote the final control location of the controller.

**Definition 2.4 (Continuous Trajectory)** *Given the actuator value* $\mathbf{u}$ *and a plant state* $\mathbf{x}_0$*, let* $\xi_{\mathbf{u}}^{\mathbf{x}_0} : \mathbb{R} \to \mathbb{R}^n$ *denote a solution to the initial value problem* $\dot{\mathbf{x}}(t) = f_{\mathbf{u}}(\mathbf{x}(t)), \mathbf{x}(0) = \mathbf{x}_0$.

Since we assumed that $f_{\mathbf{u}}(\cdot)$ is Lipschitz continuous, there exists a unique $\xi_{\mathbf{u}}^{\mathbf{x}_0}(\cdot)$ for every $\mathbf{x}_0 \in \mathbb{R}^n$.

**Definition 2.5 (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:*

- $q = ((l_1, \ldots, l_p), \mathbf{v})$, $\hat{q} = ((\hat{l}_1, \ldots, \hat{l}_p), \hat{\mathbf{v}})$, $\mathbf{y} = g(\mathbf{x})$ *denotes the sensor values corresponding to plant state* $\mathbf{x}$*, and there exists a task* $Task_j$ *such that* $\mathbf{x} = \hat{\mathbf{x}}$, $(l_j, \mathbf{v}, \hat{l}_j, \hat{\mathbf{v}}) \in \delta_j(\mathbf{y})$*, and, for every task* $Task_i$ *not equal to* $Task_j$, $l_i = \hat{l}_i$*. This is called a* controller transition.

- $q = (L_{final}, \mathbf{v})$, $\hat{q} = (L_{initial}, \mathbf{v})$, $\mathbf{u} = h(\mathbf{v})$ *denotes the actuator values corresponding to* $\mathbf{v}$, $\xi_{\mathbf{u}}^{\mathbf{x}}(t)$ *is the continuous trajectory starting from plant state* $\mathbf{x}$, *and* $\hat{\mathbf{x}} = \xi_{\mathbf{u}}^{\mathbf{x}}(t_s)$, *the value the continuous trajectory reaches at time* $t_s$. *This is called a* plant transition.

**Definition 2.6 (Trace)** *A trace of an SDCS is a finite sequence of states* $\sigma = s_0 \dots s_K$, *for some* $K$, *such that* $s_k \longrightarrow s_{k+1}$ *for all* $0 \leq k < K$.

Figure 2.2 illustrates two traces of an *SDCS*, $\sigma_a = s_0 s_1 s_2 s_3$ and $\sigma_b = s_0 s_1 s_4 s_5$. In this 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 controller variables: each plane corresponds to a different value of the controller variables, namely $\mathbf{v}_a$, $\mathbf{v}_b$, and $\mathbf{v}_c$. Plant transitions correspond to continuous lines within a given plane; controller transitions correspond to dotted lines from one plane to another. The initial state is $s_0$, and the first transition is a plant transition, $s_0 \longrightarrow s_1$. Two controller transitions are possible starting from $s_1$; nondeterminism in the controller leads to two separate states, $s_2$ and $s_4$. From each of these states a plant transition is possible, $s_2 \longrightarrow s_3$ and $s_4 \longrightarrow s_5$.

**Definition 2.7 (Duration)** *The duration of a trace* $\sigma$ *is the amount of time elapsed between its first state and its last state, and it is defined inductively as follows:*

- *if* $\sigma = s_0$, $duration(\sigma) = 0$;

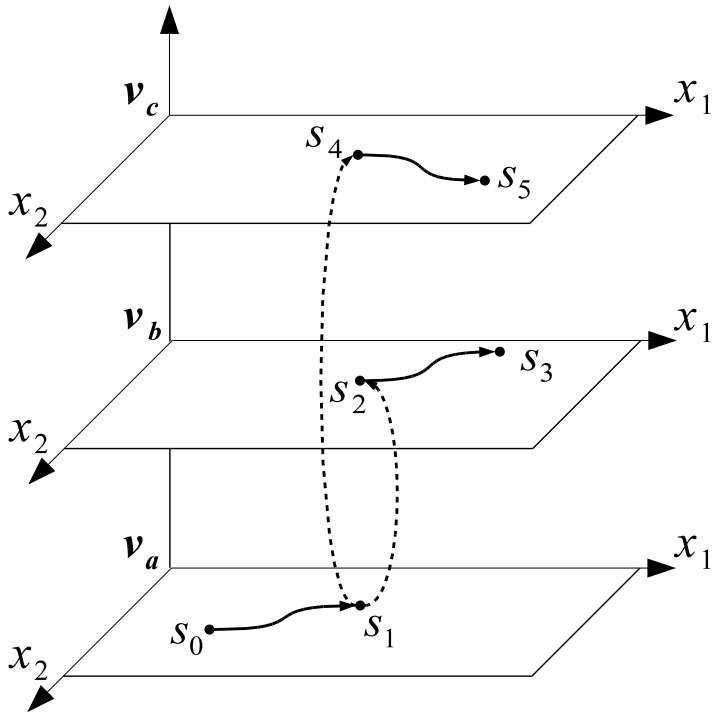



Figure 2.2: Two traces of an *SDCS*. The axes labeled $x_1$ and $x_2$ represent two dimensions of the plant state. The vertical axis represents the controller variables. Each plane correspond to a different value of the controller variables. Solid arrows connecting points represent plant transitions. Dotted lines connecting points represent controller transitions.

- *if* $\sigma = s_0 \dots s_K$ *and* $s_{K-1} \longrightarrow s_K$ *is a controller transition then* $duration(\sigma) = duration(s_0, \dots s_{K-1})$, *since we assume that controller transitions execute instantaneously;*

- *if* $\sigma = s_0 \dots s_K$ *and* $s_{K-1} \longrightarrow s_K$ *is a plant transition then* $duration(\sigma) = duration(s_0, \dots s_{K-1}) + t_s$, *since each plant transition has a duration equal to the sampling period* $t_s$.

In the model presented above, since I assumed that controller transitions are instantaneous, a plant transition is allowed only after each controller task has reached its final control location. Two situations are possible, however, in which a controller task never reaches its final control location, namely *deadlock* and *livelock*. A deadlock corresponds to a state of the controller from which no further transition is possible. A livelock corresponds to an infinite trace where the controller never reaches the control location $L_{final}$. Deadlocks and livelocks are due to errors in the controller and can be detected by the algorithms presented in this proposal. What follows are formal definitions of deadlock and livelock states.

**Definition 2.8 (Deadlock State)** *A system state $s$ is a* deadlock state *if there does not exist a system state $s'$ such that $s \longrightarrow s'$.*

Since a plant transition is always possible from the control location $L_{final}$, a deadlock corresponds to a state $(q, \mathbf{x})$ such that $L(q) \neq L_{final}$ and there does not exist a controller state $\hat{q}$ such that $(q, \hat{q}) \in \delta_i(g(\mathbf{x}))$ for some task $Task_i$, where $g(\cdot)$ is the sensor function, which maps a plant state $\mathbf{x}$ into the corresponding sensor value $\mathbf{y}$.

**Definition 2.9 (Livelock State)** *A state $s = (q, \mathbf{x})$ is a* livelock state *if and only if there exists an infinite sequence of controller states $q_0 q_1 \ldots$ such that $q_0 = q$ and, for every $i \geq 0$, there exists a task $Task_j$ such that $(q_i, q_{i+1}) \in \delta_j(g(\mathbf{x}))$, where $g(\cdot)$ is the sensor function, which maps a plant state $\mathbf{x}$ into the corresponding sensor value $\mathbf{y}$.*

**Definition 2.10 (Reachable States)** *A state $s$ of an SDCS is reachable within a time bound $T$ from a state $s_0$ if and only if there exists a trace $\sigma = s_0 \dots s_K$, for some $K$, such that $s_K = s$ and duration$(\sigma) \leq T$.*

**Definition 2.11 (Safe States)** *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 no deadlock state, no livelock state, and no state in Fail is reachable within time bound $T$.*

Consider again the diagram in Figure 2.2: states $s_3$ and $s_5$ are safe for time bound zero, states $s_1$, $s_2$, and $s_4$ are safe for time bound $t_s$, and state $s_0$ is safe for time bound $2t_s$.

**Definition 2.12 (Bounded-Time Safety)** *Given an SDCS, a set of states Fail $\subset Loc \times V^m \times \mathbb{R}^n$, 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 $T$.*

# Chapter 3

# Systematic Simulation

Numerical simulation is a validation technique that generates a trace of a system. For a dynamical system specified as a set of differential equations, numerical methods are used to perform the simulation. Tools such as MATLAB/Simulink [1] 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, for

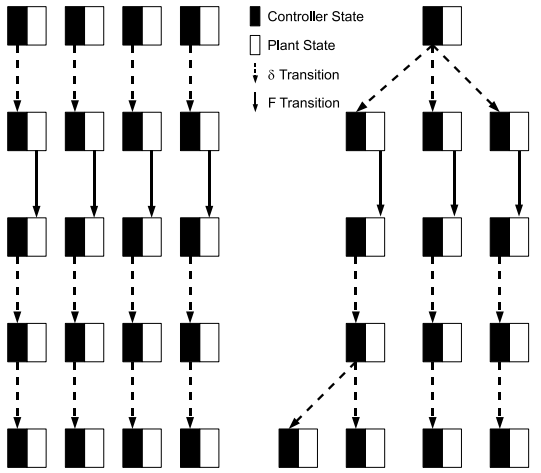
example, external inputs to the controller.

In contrast to numerical simulation, where each trace is explored independently, in model checking the set of generated traces forms a graph, known as the reachable state-space graph (see Figure 3.1). The nodes of the graph correspond to states of the system, the edges to transitions. A state of the graph is initial if the corresponding system state is initial. A path in the graph from an initial state corresponds to a trace of the system. Explicit-state model checkers explore the reachable state-space graph starting from each element of a finite set of initial states. The algorithm is based on a depth first search of the state-space graph. The search proceeds from a state to one of its successors and continues until it reaches a state that has no successor or a previously visited state. This approach guarantees that no transition is visited more than once. In terms of the traces that need to be explored, this leads to a saving in terms of simulation time, as sequences of states that are common to multiple traces are explored only once.

## 3.1 Algorithm

The algorithm used by our approach is shown in Figure 3.2. The `main` function takes as inputs an *SDCS*, a set of fail states *Fail*, and a time bound $T$. It returns one of four possible answers:

- **SAFE**, if the system is safe within time bound $T$ and no deadlock or livelock is reachable within time bound $T$;




(a) Standard simulation (b) Systematic simulation

Figure 3.1: Standard simulation (a) generates traces one at a time. Systematic simulation (b) exploits the common prefixes of traces to make the analysis more efficient.

- **UNSAFE**($\sigma$), if the system is unsafe; $\sigma$ is a trace that leads to an unsafe state;
- **DEADLOCK**($\sigma$), if a deadlock state is reachable within $T$; $\sigma$ is a trace that leads to a deadlock state;
- **LIVELOCK**($\sigma$), if a livelock state is reachable within $T$; $\sigma$ is a trace that leads to a livelock state.

The main function calls the function `explore` for each initial state of the *SDCS* (lines 5-6) and appropriately interprets the result (lines 7-8). The function `explore` takes as arguments the system state $(q, \mathbf{x})$, the time horizon $\tau$, and the sequence of states $\sigma$. This function performs a depth-first search

of the part of the state-space graph reachable from $(q, \mathbf{x})$ up to time $\tau$.

```
1:  global SDCS, Fail, T;
2:  global visited ← {}; // Visited states, initially empty.

3:   // Check the time-bound safety of an SDCS
4:  main :
5:    foreach ((q, x) ∈ Init) : // Depth-first search for each initial state
6:      result ← explore(q, x, T, ());
7:      if(result != SAFE) return result;
8:    return SAFE;

9:   // Perform a depth-first search up to time τ
10: function explore(q, x, τ, σ) :
11:   if ((q, x) ∈ Fail) return UNSAFE(σ); // Check for fail states
12:   if (is_deadlock(q, x)) return DEADLOCK(σ) // Detect deadlocks
13:   if (is_livelock(q, x, σ)) return LIVELOCK(σ) // Detect livelocks

14:    // Compare to already visited states
15:   if (already_visited(q, x, τ))
16:     return SAFE;
17:   visited ← visited ∪ {(q, x, τ)};

18:   if (q.L = L_final) : // Perform a plant transition
19:     return plant_transition(q, x, τ, σ);
20:   else : // Perform a controller transition
21:     return controller_transitions(q, x, τ, σ);
```

Figure 3.2: The systematic simulation algorithm.

The function `explore` checks if an unsafe state has been reached (line 11), in which case it returns immediately. Otherwise, `explore` checks for deadlocks (line 12) and livelocks (line 13). Next, if there exists an already visited state that is equal to the current state and has a time horizon larger than or equal to the time horizon of the current state (line 15 and function

```
22:  // Perform a plant transition using numerical simulation
23: function plant_transition(q, x, τ, σ) :
24:    if (τ < t_s) return SAFE;  // Stop if not enough time left
25:    x̂ ← sim(x, h(q.v));
26:    q̂ ← (L_initial, q.v);
27:    return explore(q̂, x̂, τ − t_s, σ · (q̂, x̂));

28:  // Perform all possible controller transitions
29: function controller_transitions(q, x, τ, σ) :
30:    Q̂ ← {q̂ | ∃ i : (q, q̂) ∈ δ_i(g(x))} ;
31:    foreach (q̂ ∈ Q̂)
32:       result ← explore(q̂, x, τ, σ · (q̂, x));
33:       if (result != SAFE) return result;
34:    return SAFE;

35:  // Check for deadlocks
36: function is_deadlock(q, x) :
37:    Q̂ ← {q̂ | ∃ i : (q, q̂) ∈ δ_i(g(x))} ;
38:    return (Q̂ = ∅);

39:  // Check for livelocks
40: function is_livelock(q, x, σ) :
41:    K = σ.length;
42:     // Check if there exists a loop made of only discrete transitions.
43:    if (∃ N < K :
44:          σ[N] = (q, x) ∧
45:          ∀ N ≤ k < K :
46:             ∃ i : (σ[k].q, σ[k + 1].q) ∈ δ_i(σ[k].x) ∧
47:             σ[k].x = σ[k + 1].x) :
48:       return true;
49:    return false;

50:  // Check the state has already been visited
51: function already_visited(q, x, τ) :
52:    return (∃ (q̂, x̂, τ̂) ∈ visited : q = q̂ ∧ x = x̂ ∧ τ ≤ τ̂);
```

Figure 3.2: Continued – The systematic simulation algorithm.

`already_visited` at lines 51-52), the search stops as the current state is guaranteed to be safe (line 16). Otherwise, the current state and time horizon are added to the set of visited states (line 17).

Line 19 calls the function `plant_transition` to perform a plant transition and is executed only if the current control location $q.L$ is equal to $L_{final}$. Otherwise line 21 calls the function `supervisor_transitions`, which explores the transitions of the controller.

The function `plant_transition` first checks that the time horizon is large enough to allow a plant transition, whose duration is equal to the sampling time $t_s$ (line 24). If a transition is possible, lines 25-26 compute the next state: the plant state $\hat{\mathbf{x}}$ is obtained by performing numerical simulation (calling the function `sim`) and the control location is set to $L_{initial}$. The exploration continues on line 27 with a recursive call to `explore` starting from the newly generated state, with a smaller time horizon and an updated trace.

The function `controller_transitions` first computes the set of successors of the given state (line 30). The successors are explored, one at a time, by the loop at lines 31-33. For each successor, a recursive call to `explore` is performed. The time horizon in the recursive call is unchanged since I assumed that controller transitions execute instantaneously (line 34).

The function `is_deadlock` (lines 36-38) determines if the current state is a deadlock by looking at the set of its successors and checking whether it is empty. The function `is_livelock` (lines 40-49) determines if the current state belongs to a livelock by checking whether it is part of a loop in the

current path $\sigma$. The function `already_visited` (lines 51-52) determines if the current state has already been visited by looking for a state with the same controller state $q$, the same plant state $\mathbf{x}$, and a time horizon greater than or equal to the current time horizon $\tau$.

The pseudocode in Figure 3.2 is based on the algorithm for explicit-state model checking [9], which performs a depth-first search of the state transition graph. The major additions are *livelock detection* (lines 44-49), bounded time reachability by storing *time horizons* together with states in the visited set (lines 15-17), and the computation of *plant transitions* using numerical simulation (lines 23-27).

## 3.2 Implementation

We implemented this technique by extending an existing explicit-state source code model checker [21]. The tool we chose to use is Java PathFinder [27]. 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 code generator with only minor syntactic modifications. The main reasons for choosing Java PathFinder 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. This enables full support of models developed used MATLAB/Simulink, a standard modeling tool for these types of systems.

We extended the existing model checker in the following ways. 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 plant 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. Plant transitions are computed using the MATLAB/Simulink numerical integration solver (this corresponds to line 31 in Figure 3.2). Given the sampling period $t_s$, the current plant state $\mathbf{x}$, and the actuator values $\mathbf{u}$, MATLAB/Simulink returns the state $\hat{\mathbf{x}}$ that is reached at time $t_s$.

## 3.3 Experimental Evaluation

In the following, I present experimental results obtained by applying the systematic simulation technique presented above 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 [15]. We obtained a Simulink model of the STARMAC system from the Stanford development team. Then we constructed a new system model, the Reconnaissance Mission (RM) model, that

includes a supervisory controller that we designed. We used the technique described above to detect an error in the RM model.

The vehicle, shown in Figure 3.3, 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 3.4. 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.

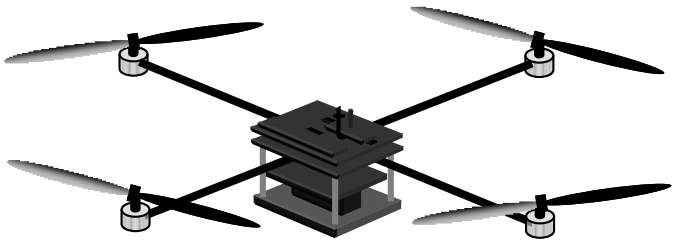


Figure 3.3: An illustration of the Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control.

We constructed a supervisory controller whose purpose is to guide the vehicle through a sequence of waypoints. 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 super-

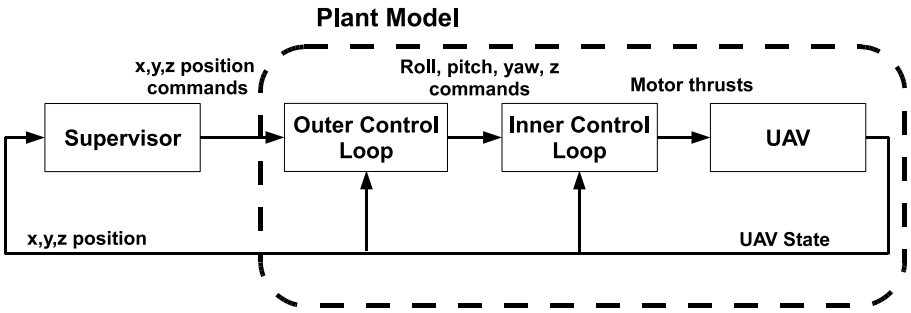



Figure 3.4: Vehicle block diagram.

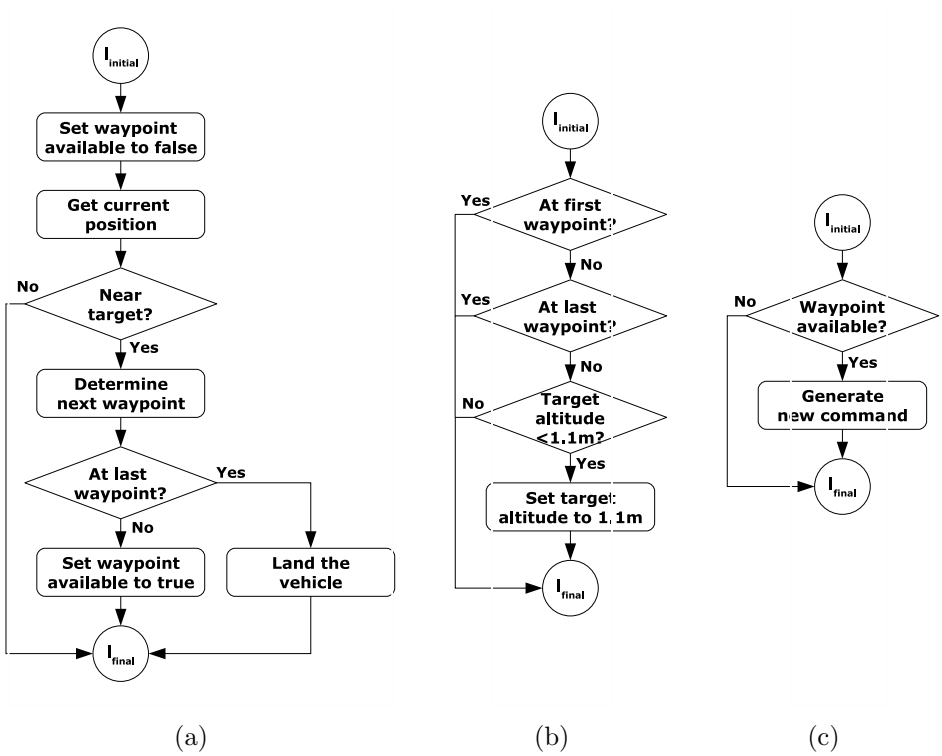
visory controller is modeled using Stateflow diagrams. The implementation uses the following interleaved tasks, illustrated in Figure 3.5:

- **Waypoint Tracking** [Figure 3.5(a)] 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 picks the next waypoint from the list and issues the command to the STARMAC Quadrotor.

- **Waypoint Monitor** [Figure 3.5(b)] 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 to avoid falling below the 1 meter minimum altitude.

- **Command Latch** [Figure 3.5(c)] maintains the last command until the next waypoint command is issued.

The MATLAB/Simulink model of the RM system includes the supervi-




(a) (b) (c)

Figure 3.5: The three tasks implementing the controller: (a) Waypoint Tracking; (b) Waypoint Monitor; and (c) Command Latch.

sory controller, the outer loop controller, the inner loop controller, and the dynamics of the vehicle (see Figure 3.4). 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 to be checked 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 this chapter 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 3.6 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.

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 3.7). 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

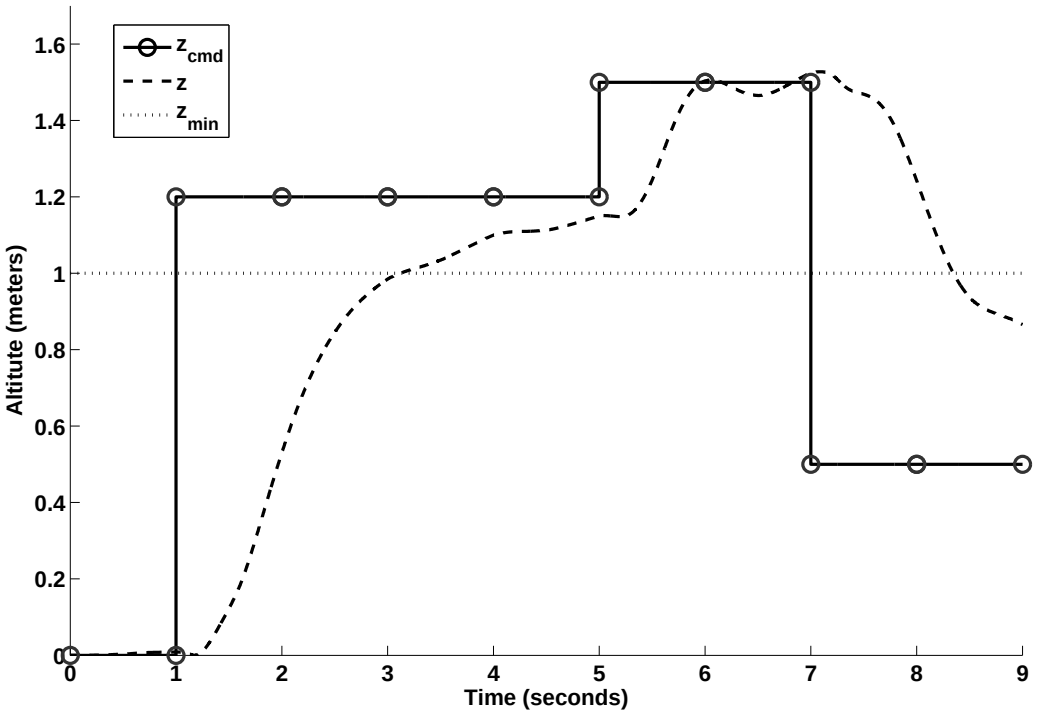



Figure 3.6: Counterexample trace.

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 the tool.

As shown in Figure 3.8, 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 3.6 contains 1346 transitions, of which 9 are plant transitions and

| # | 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 |

Figure 3.7: List of waypoints used in the experimental evaluation

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 equal to previously visited states, marked as revisited states in the table. 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 3.8. 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.

Figure 3.8: Running times, memory usage, number of reached states, and number of revisited states for different time bounds and with and without approximate equivalence.

# Chapter 4

# Path Pruning

The systematic simulation approach presented in the previous chapter 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. This chapter presents an approach that prunes the state-space graph by removing parts of its traces.

Explicit-state model checkers compute the set of reachable states iteratively by constructing the state-space graph using the transition relation of the system. When the model checker encounters a state that is equal to a previously visited state (lines 15-16 in Figure 3.2), the successors of the

current state are not explored. Doing so would only lead to states that have already been encountered. This chapter presents an approach that replaces the notion of state equality in model checking with state equivalence based on an approximation of the plant state. The approach, called *path pruning*, uses a notion of *approximate equivalence* between states. This approach, however, is a heuristic 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 cannot prove that a system is safe.

## 4.1 Approximate Equivalence

The path pruning approach presented in this chapter is based on a notion of *approximate equivalence* between states. Intuitively, an approximate equivalence corresponds to an equivalence relation between the plant states.

**Definition 4.1 (Approximate Equivalence)** *Given an SDCS, an equivalence relation $\approx$ between plant states, two states $(q_1, \mathbf{x}_1)$ and $(q_2, \mathbf{x}_2)$ of the SDCS are approximately equivalent if and only if $q_1 = q_2$ and $\mathbf{x}_1 \approx \mathbf{x}_2$.*

The notion of approximate equivalence defined above is used in the next section to prune the paths explored by the algorithm presented in the previous chapter. When a state that is approximately equivalent to a previously visited state is encountered, we can stop exploring traces starting from that

state. In the following, I present two examples of equivalence relations over the plant states.

**Example 4.1 (Finite Precision)** The first equivalence relation is obtained using a finite precision to represent each component of the plant state. Given a finite precision $\delta > 0$, the finite precision representation of a real number is given by the function $fp : \mathbb{R} \to \mathbb{R}$ defined as $fp_\delta(r) = \delta\lfloor r/\delta \rfloor$. For example, if $r = 3.259$ and $\delta = 0.1$, we have that $fp_\delta(r) = 3.2$. Given a plant state $\mathbf{x} \in \mathbb{R}^n$, let $\mathbf{x}|_i$ denote the $i^{\text{th}}$ component of $\mathbf{x}$. Given a finite precision $\delta > 0$, the $\delta$-approximate equivalence relation $\approx_\delta$ is defined as follows: two plant states $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$ are equivalent, denoted by $\mathbf{x}_1 \approx_\delta \mathbf{x}_2$, if and only if, for every $i$ between 1 and $n$, $fp_\delta(\mathbf{x}_1|_i) = fp_\delta(\mathbf{x}_2|_i)$. The relation $\approx_\delta$ is clearly an equivalence relation. This notion of approximate equivalence corresponds to pruning the paths starting from a state that is equal to a previously visited state when using only a finite precision to compare the plant states. Figure 4.1(a) shows two plant states $\mathbf{x}_1$ and $\mathbf{x}_2$ that are approximately equivalent. The hatched area represents the set of states equivalent to $\mathbf{x}_1$ and $\mathbf{x}_2$ according to this notion of equivalence.

**Example 4.2 (Order Reduction)** A different equivalence relation between plant states can be defined by considering only some components of the plant state. Let $\Pi : \mathbb{R}^n \to \mathbb{R}^{n'}$, with $n' < n$, denote the projection of a subset of components of a plant state $\mathbf{x} \in \mathbb{R}^n$. The $\Pi$-approximate equivalence $\approx_\Pi$ is defined as follows: two plant states $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$ are equivalent if and only if

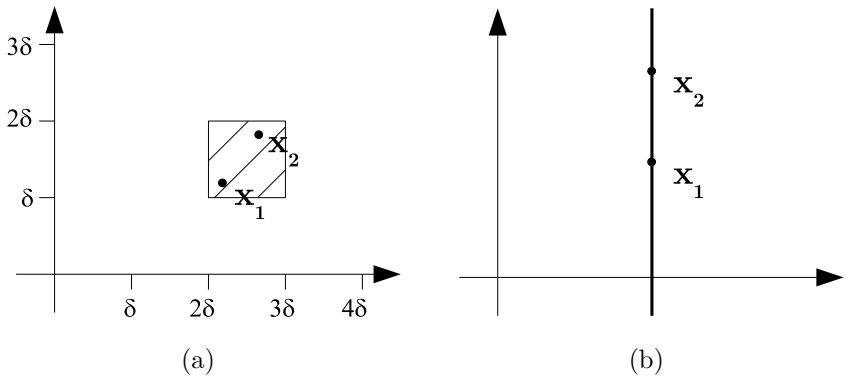



Figure 4.1: Two equivalent plant states $\mathbf{x}_1$ and $\mathbf{x}_2$ using different notions of equivalence: (a) Finite precision; and (b) Order reduction.

$\Pi(\mathbf{x}_1) = \Pi(\mathbf{x}_2)$. It is easy to see that $\approx_\Pi$ is an equivalence relation. This notion of equivalence corresponds to pruning traces starting from a state that is equal to a previously visited state with respect to a subset of its components. Figure 4.1(b) shows two equivalent plant states $\mathbf{x}_1$ and $\mathbf{x}_2$. Using this notion of equivalence, the set of states equivalent to $\mathbf{x}_1$ and $\mathbf{x}_2$ corresponds to a vertical line.

## 4.2 Path Pruning Algorithm

By using the notion of approximate equivalence defined in the previous section it is possible to modify the algorithm given in the Chapter 3 (Figure 3.2) to search for counterexamples efficiently. Notice that by using approximate equivalence, the algorithm is unable to prove bounded-time safety. However, it allows to search efficiently for counterexample in a complex system by

looking only at a subset of its traces. The next chapter presents an extension of this approach and introduces a notion of equivalence that is sufficient to guarantee safety.

The *path pruning algorithm* is a simple modification of the systematic simulation algorithm presented in Chapter 3 where the `already_visited` function defined in Figure 3.2 is replaced by the one shown in Figure 4.3. The algorithm simply checks if a state that is approximately equivalent to the one currently being explored has been visited before. If such a state exists then the successors of the current states are not explored. I call this operation *path pruning* as it corresponds to removing the paths starting at the current state from the state-space graph constructed by the algorithm. Figure 4.2(a) shows an example of path pruning.

## 4.3 Experimental Evaluation

We have implemented the path pruning approach presented above in a model checker based on Java PathFinder [25]. In this implementation and in the example below, we have used the finite precision approach from Example 4.1 to compute approximate equivalence. Given a precision $\delta > 0$, the $\delta$-approximate equivalence relation $\approx_\delta$ corresponds to using only a finite precision to represent the states of the plant. Therefore, instead of storing the plant state in the `visited` set (line 17 in Figure 3.2), we decided to store the finite precision approximation of the plant state instead. This has some

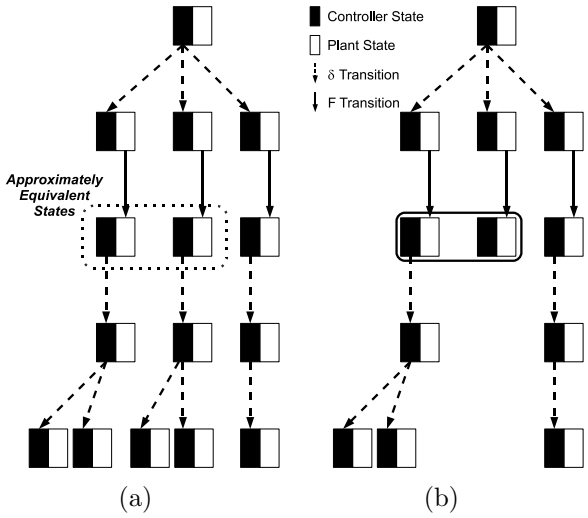



Figure 4.2: Approximate equivalence (a) identifies states that have the same controller state and similar plant states. This enables pruning (b) parts of the state-space graph when checking for approximate safety.

practical advantages. First, in order to determine if a state is approximately equivalent to the current state, it is sufficient to find a stored state that is equal to the finite precision approximation of the current state. This allows using efficient hashing techniques to determine if a state has already been visited. Another advantage is that storing the finite precision approximation of states reduces the memory requirements needs to represent the set of visited states.

Below I demonstrate the path pruning algorithm by applying it to a robotic example. I first describe the robotic example itself. After that, I outline the steps involved in analyzing the system and the results I obtained.

A slalom course laid out on the pavement defines the task for the robot.

```
1:  // Check if the state has already been visited
2: function already_visited(q, x, τ) :
3:   return
4:         ∃ (q̂, x̂, τ̂) ∈ visited :
5:           q = q̂ ∧
6:           x ≈ x̂ ∧
7:           τ ≤ τ̂;
```

Figure 4.3: The conservative merging verification algorithm.

A white line connects a series of gates which the robot must follow while controlling the speed. Various weather and lighting conditions challenge sensing, computing, and locomotion. The robot tries to complete the course as quickly as possible while tracking the line. A robot designed to meet the specification requires an adequate physical platform with correct software.

**Hardware.** The robot is 41x27x12cm in size and utilizes two battery packs with 7.2V and 12V to provide power for processing and actuators independently. Two geared 12V motors drive the back wheels as depicted in Fig. 4.4. The microcontroller adjusts the pulsewidth of the signal and effectively regulates the power supplied to the motors. To actuate the rack-and-pinion steering, the microcontroller sends pulsewidth commands to a servo. Steering and velocity commands are determined using the data read from the sensors. The robot has 12 sensors (Fig. 4.4): ten brightness sensors measure the reflectivity of the ground and two encoders measure the velocity and distance traveled. The brightness sensors are mounted on a line perpendicular to the direction of travel. The approximate offset from the center of the line

is measured using this sensor arrangement. The input/output (I/O) ports of the microcontroller are connected to an analog-to-digital (A/D) converter, 25-tick encoders, a servo, and an amplifier. Three tasks read and write values via I/O ports and memory. The microcontroller directly executes Java, which significantly simplifies the verification using Java PathFinder. It implements Java 2 Micro Edition, and it has multithreading. The software is written and compiled on a host computer and then downloaded to the flash memory of the microcontroller.

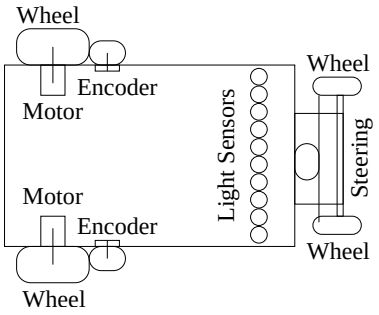



Figure 4.4: Diagram of the robot showing the location of sensors and actuators.

**Software.** The software regulates the steering and the speed using two separate tasks. An additional task reads the reflectance values from the brightness sensors. Computations do not involve dynamic object creation or destruction and use only integer arithmetic. Depending on the setup, different interleaving of tasks are possible but all tasks execute periodically with a frequency of 33Hz. The A/D converter measures the voltage generated by each light sensor. The microcontroller communicates via a serial bus to

read the values. Since commands are sent at the fixed frequency of 33Hz, sensor values are read at the same frequency. The two encoders are connected directly to the microcontroller and provide speed and distance traveled. The cached brightness values are used by the controller to calculate a steering command. After normalizing the values, the algorithm finds the left and right edge of the line. Proportional control is performed to try to keep the middle of the line centered underneath the sensors. If it is ever detected that the line is out of bounds, the last known good direction is used as the steering direction. The speed of the robot is calculated from elapsed time and registered encoder ticks. The desired goal speed is determined from the the last steering command in order to adapt to curvature. The difference between the two determines the pulsewidth command sent to the motor driver.

The system is depicted in a block diagram representation in Fig. 4.5. The control software is on the left and the plant is on the right. The inputs and outputs of the controller software are quantized since the implementation of the software relies on fixed point integer arithmetic. The dependency between the steering and speed control tasks is indicated by a connection from the output of the steering controller to the input of the speed controller.

**The Plant Model.** The plant model can be either derived from first principles or automatically identified using a technique known as *system identification* [23]. Although the robot is simple, it is not trivial to derive the physical model from first principles because many parameters are unknown

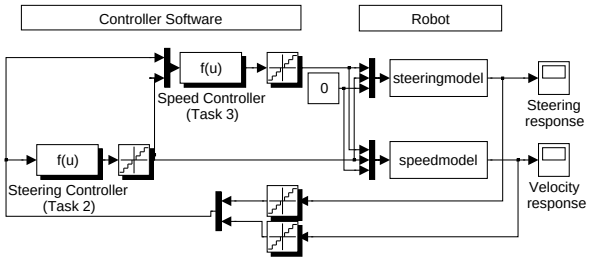



Figure 4.5: A block diagram representing the control software and discrete time plant model of the robot.

and hard to measure. Therefore, we derived the environment using system identification. The position relative to the line depends on the velocity of the robot and the steering angle. A fourth-order system gives a good relationship between velocity, steering pulsewidth, and position relative to the line. The identified system model constrains the input into the software. Since the interactions with the environment are executed at a fixed frequency of 33Hz, we express the system as a discrete state space systems with a sample time of $0.03s$.

**Properties.** The initial configuration of the robot defines the initial values of the state vector. Although it is possible to test ranges of initial configurations, we focused on a single initial configuration. In this configuration the robot steers to the right since it wants to reach back to the center of the line. The robot is offset from the middle of the line by 36mm, i.e. the middle of the line is under the rightmost sensor. The robot starts with zero initial velocity. The closed system with initial conditions is fully specified

and permits reasoning about the properties that are necessary to ensure the correct operation of the robot. Safety properties state conditions that must hold for the robot to be able to advance and not damage the hardware. In particular, the line has to be visible by at least one light sensor to ensure that a correct command is sent. This is specified as a property which checks that the position with respect to the line is within a given range. The speed also has to be within a range to give the robot enough time to process and react to the available data.

The goal of the case study is to demonstrate the path pruning approach to verifying control system software. By using the path pruning algorithm presented in this chapter, we were able to check approximate safety of the system (Figure 4.6). In order to examine the effects of different kinds of sensor and actuator inaccuracy, we added nondeterministic behavior to parts of the model. We explored the influence of a $\pm 5$ error in the number of ticks counted by the encoder, a random failure of two light sensors, and a $\pm 50 \mu s$ error in the output pulsewidth times. The runtime and memory usage for the different runs are shown in Figure 4.6. While modeling errors requires additional resources, it provides a better coverage of the behavior of the system.

In order to determine if the approach is able to detect subtle errors in the code, we seeded a bug in controller. In [22], a fatal type-conversion bug for the Ariane 5 rocket is analyzed. In this incident, the conversion of a 64-bit floating point value to a 16-bit signed integer caused an overflow.

| **Experiment** | **Time** | **Memory** |
|---|---|---|
| Model without Noise | 0:00:07 | 5 Mb |
| Noisy Encoder Values | 0:45:41 | 106 Mb |
| Light Sensor Failure | 1:12:55 | 159 Mb |
| Noisy Pulsewidth Commands | 0:18:18 | 41 Mb |

Figure 4.6: Computation time and memory necessary for the verification with and without different kinds of errors.

This occurs only for specific trajectories, making it a perfect candidate for model checking. We investigated if this type of bug could be discovered using our technique. The source code of the robot was seeded with a similar type-conversion bug. An 8-bit signed integer variable was used to store the number of encoder ticks during a single period. The internal value is actually a 16-bit unsigned integer, so a type conversion occurs when the value is read from the sensor. The 8-bit signed integer variable, which ranges between -127 and 128, is sufficient in most cases to represent the number of ticks, as it is usually less than 128. However, if the velocity exceeds a certain threshold, a number of ticks larger than 128 is possible. The bug is hard to find, as it depends on the continuous state of the system (the current velocity) and the bug shows itself only under certain conditions (the velocity exceeding the threshold). Even if it is in general difficult to detect intermittent bugs, we were able to detech this bug using our approach.

# Chapter 5

# Conservative Merging

In Chapter 3, I presented an approach that combines model checking and simulation to check bounded-time safety of an *SDCS* with a finite set of initial states. In Chapter 4 I introduced 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 not guaranteed to find a counterexample if the system is not safe.

The approach presented in this chapter is able to prove bounded-time safety of an *SDCS* while performing path pruning, which corresponds to merging a trace 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 *conservative merging* 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, I show how to determine safe sets of plant states around the states 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 with a previously visited state 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 thesis 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 [11]. In this work, we use bisimulation functions to approximate conservatively the plant transitions.

**Definition 5.1 (Bisimilation Function)** [11] *Given an autonomous dynamical system* $\Sigma$ *described by* $\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$ *where* $\mathbf{x} : \mathbb{R} \rightarrow \mathbb{R}^n$ *and* $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$, *a differentiable function* $\varphi : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$ *is a bisimulation function of* $\Sigma$ *if and only if*

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

The previous definition is similar to the definition of Lyapunov functions from stability theory [6]. Both Lyapunov functions and bisimulation functions are required to decrease along trajectories of the system (the second

condition of Definition 5.1). Bisimulation functions, however, are defined over a pair of states and consider the distance between two trajectories, while Lyapunov functions are defined over a single state and consider the distance between a trajectory and a rest post. Lyapunov functions can be used to prove that a trajectory starting close to a rest point will remain close to the rest point as time evolves. Similarly, bisimulation functions can be used to prove that two trajectories that start from points close to each other remain close to each other as time evolves [16].

**Definition 5.2 (Inner Levels Sets)** *Given a plant state* $\mathbf{x} \in \mathbb{R}^n$*, a bisimulation function* $\varphi$*, and a real value* $r \geq 0$*, the* inner levels set *of the bisimulation function* $\varphi$ *centered at* $\mathbf{x}$ *and of size* $r$*, denoted by* $\mathcal{N}_\varphi(\mathbf{x}, r)$*, is defined as*

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

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. [16].

**Theorem 5.1 (Plant Approximation)** *Given two states* $s = ((L_{final}, \mathbf{v}), \mathbf{y})$ *and* $\hat{s} = ((L_{initial}, \mathbf{v}), \hat{\mathbf{y}})$ *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* $((L_{final}, \mathbf{v}), \mathbf{z}) \longrightarrow ((L_{initial}, \mathbf{v}), \hat{\mathbf{z}})$ *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 [16]. □

In Chapter 2, I defined a state of an SDCS to be safe if it does not lead to a fail state, a deadlock state, or a livelock state (see Definition 2.11). A state of an SDCS is made of a supervisor state $q$ and a plant state $\mathbf{x}$. If I choose a specific supervisor state $q$, I can define a set $\mathcal{X}$ of plant state to be safe with respect to the supervisor state $q$ as follows.

**Definition 5.3 (Safe Plant States)** *Given a supervisor state $q$ and a time bound $T$, a set $\mathcal{X} \subseteq \mathbb{R}^n$ of plant states is safe for $T$ at $q$ if and only if, for every $\mathbf{x} \in \mathcal{X}$, the state $(q, \mathbf{x})$ is safe for time bound $T$.*

Figure 5.1 illustrates the notion of safe plant states. In this 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. 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 only for

one sampling period. The set $N_1$ is also safe for time bound $t_s$ as discrete transitions from states in $N_5$ lead to states in $N_1$ or in $N_3$.

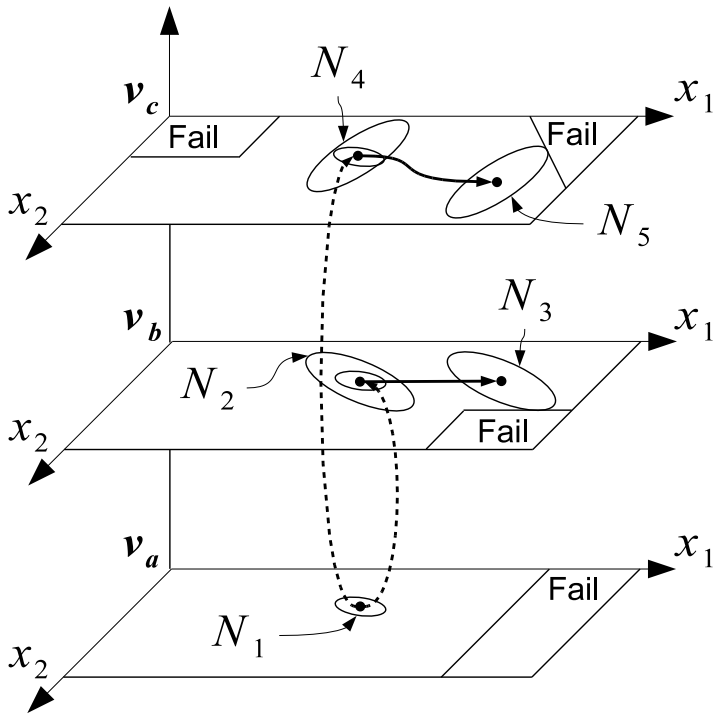



Figure 5.1: An illustration of sets of plant states safe for a time bound $T$. $N_5$ and $N_3$ are safe for time bound zero, while $N_1$, $N_2$ and $N_4$ are safe for time bound $t_s$, the sampling period.

Given a set of fail states *Fail* and a supervisor state $q$, let us denote by $Fail_q$ the *fail states of* $q$, i.e., the set of plant states that, together with supervisor state $q$, are fail states. The set of fail states of $q$ can be defined as

$$Fail_q = \{\mathbf{x} \in \mathbb{R}^n \mid (q, \mathbf{x}) \in Fail\}\,.$$

**Theorem 5.2 (Plant Transition Approximation)** *Given two states* $(q, \mathbf{y})$ *and* $(\hat{q}, \hat{\mathbf{y}})$ *such that* $q = (L_{final}, \mathbf{v})$, $\hat{q} = (L_{initial}, \mathbf{v})$, *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* $\mathcal{N}_{\varphi_\mathbf{v}}(\mathbf{y}, r) \cap Fail_q = \emptyset$ *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 $\sigma = s_0 s_1 \ldots s_K$, for some $K$, such that $s_0 = (q, \mathbf{z})$, $s_K \in Fail$, and $duration(\sigma) \leq T + t_s$. Since $\mathbf{z} \in \mathcal{N}_{\varphi_\mathbf{v}}(\mathbf{y}, r)$ and, by hypothesis, $\mathcal{N}_{\varphi_\mathbf{v}}(\mathbf{y}, r)$ does not intersect the fail states of $q$, $\mathcal{N}_{\varphi_\mathbf{v}}(\mathbf{y}, r) \cap Fail_q = \emptyset$, we have that $\mathbf{z} \notin Fail_q$ and so $s_0 = (q, \mathbf{z}) \notin Fail$. Therefore, the trace must contain at least two states ($K \geq 1$). Let $\hat{\sigma}$ denote $s_1 \ldots s_K$. By Definition 2.7 we have that $duration(\sigma) = duration(\hat{\sigma}) + t_s$, since $\hat{\sigma}$ is obtained from $\sigma$ by removing a plant transition, whose duration is the sampling period $t_s$. Since $duration(\sigma) \leq T + t_s$, by assumption, we can deduce that $duration(\hat{\sigma}) \leq T$. By Definition 2.5, $s_1 = (\hat{q}, \hat{\mathbf{z}})$ for some $\hat{\mathbf{z}} \in \mathbb{R}^n$. By Theorem 5.1 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 exist 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 5.2 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 supervisor transitions. While plant 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 plant states with respect to a supervisor state.

**Definition 5.4 (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 and only if*

- *for every $\hat{q_1}$ such that $(q, \mathbf{y}) \longrightarrow (\hat{q_1}, \mathbf{y})$, we have that $(q, \mathbf{z}) \longrightarrow (\hat{q_1}, \mathbf{z})$; and*
- *for every $\hat{q_2}$ such that $(q, \mathbf{z}) \longrightarrow (\hat{q_2}, \mathbf{z})$, we have that $(q, \mathbf{y}) \longrightarrow (\hat{q_2}, \mathbf{y})$.*

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 $[\mathbf{y}]_q$ denote the equivalence class of $\mathbf{y}$ defined by $\approx_q$, that is $[\mathbf{y}]_q = \{\mathbf{z} \in \mathbb{R}^n \mid \mathbf{y} \approx_q \mathbf{z}\}$.

**Theorem 5.3 (Supervisor Transition Approximation)** *Given a state $(q, \mathbf{y})$ with $q = (L, \mathbf{v})$ and $L \neq L_{final}$, let $\hat{Q} = \{\hat{q} \mid (q, \mathbf{y}) \longrightarrow (\hat{q}, \mathbf{y})\}$ denote the set of supervisor successors of $(q, \mathbf{y})$. For each successor $\hat{q} \in \hat{Q}$, let $T_{\hat{q}}$ be a time*

*bound and* $\hat{\mathcal{X}}_{\hat{q}} \subseteq \mathbb{R}^n$ *be a set of plant states safe for time bound* $T_{\hat{q}}$ *at* $\hat{q}$*. Consider a time bound* $T$ *and set of plant states* $\mathcal{X} \subseteq \mathbb{R}^n$*. The set* $\mathcal{X}$ *is safe for time bound* $T$ *if*

- *for every supervisor successor* $\hat{q} \in \hat{Q}$*, the time bound* $T$ *is less than the time bound* $T_{\hat{q}}$ *corresponding to* $\hat{q}$*, i.e.,* $T \leq T_{\hat{q}}$*;*
- *the set* $\mathcal{X}$ *does not contain any fail states of* $q$*, i.e.,* $\mathcal{X} \cap \mathit{Fail}_q = \emptyset$*;*
- *every plant state in* $\mathcal{X}$ *is program equivalent to* $\mathbf{y}$*, i.e.,* $\mathcal{X} \subseteq [\mathbf{y}]_q$*; and*
- *for every supervisor successor* $\hat{q} \in \hat{Q}$*, every plant state in* $\mathcal{X}$ *belongs to the set of safe plant states* $\hat{\mathcal{X}}_{\hat{q}}$ *corresponding to* $\hat{q}$*, i.e.,* $\mathcal{X} \subseteq \hat{\mathcal{X}}_{\hat{q}}$*.*

**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 $s_0 = (q, \mathbf{z})$, $s_K \in \mathit{Fail}$, and $\mathit{duration}(\sigma) \leq T$. Since $\mathbf{z} \in \mathcal{X}$ and, by hypothesis, $\mathcal{X}$ does not intersect the fail states of $q$, $\mathcal{X} \cap \mathit{Fail}_q = \emptyset$, we know that $\mathbf{z} \notin \mathit{Fail}_q$ and so $s_0 = (q, \mathbf{z}) \notin \mathit{Fail}$. Therefore the trace $\sigma$ must contain at least two states ($K \geq 1$). The first transition of $\sigma$ must be a supervisor transition because $L \neq L_{\mathit{final}}$ by hypothesis. Let $s_1 = (\hat{q}, \mathbf{z})$ and $\hat{\sigma} = s_1 \ldots s_K$. Since $\mathbf{z} \in \mathcal{X}$ and, by hypothesis, $\mathcal{X} \subseteq [\mathbf{y}]_q$, we have that $\mathbf{y} \approx_q \mathbf{z}$ and, since $(q, \mathbf{z}) \longrightarrow (\hat{q}, \mathbf{z})$, there exists a supervisor transition $(q, \mathbf{y}) \longrightarrow (\hat{q}, \mathbf{y})$ and so it must be that $\hat{q} \in \hat{Q}$. Since we assumed that $\mathbf{z} \in \mathcal{X}$ and, by hypothesis, $\mathcal{X} \subseteq \hat{\mathcal{X}}_{\hat{q}}$ because $\hat{q} \in \hat{Q}$, 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 $\mathit{duration}(\hat{\sigma}) \leq T_{\hat{q}}$. This is true because $\hat{\sigma}$ is obtained from $\sigma$ by removing a supervisor transition and therefore $\mathit{duration}(\sigma) = \mathit{duration}(\hat{\sigma})$, $\mathit{duration}(\sigma) \leq T$, by assumption, and $T \leq T_{\hat{q}}$, by hypothesis. This is a contradiction and therefore $\mathcal{X}$ must be safe for $T$ at $q$. □

Figure 5.2 shows an application of the theorem above. In this case, 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 $T_2$ and $T_4$, respectively. The set $X_1$ in Figure 5.2 denotes the equivalence class $[\mathbf{y}]_q$ corresponding to state $s_1$. Let $T_1$ be the smallest of $T_2$ and $T_4$. Then, set $N_1$ is safe for $T_1$, 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 5.3 is within $N_2$, which we assume to be safe for time bound $T_2$. The state $s_6$ has a single successor, namely $s_7$. The set $X_6$ in Figure 5.3 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 5.3, we can deduce that $N_6$ is safe for $T_2$: any trace starting from a plant state within $N_6$ leads to a state within $N_2$.

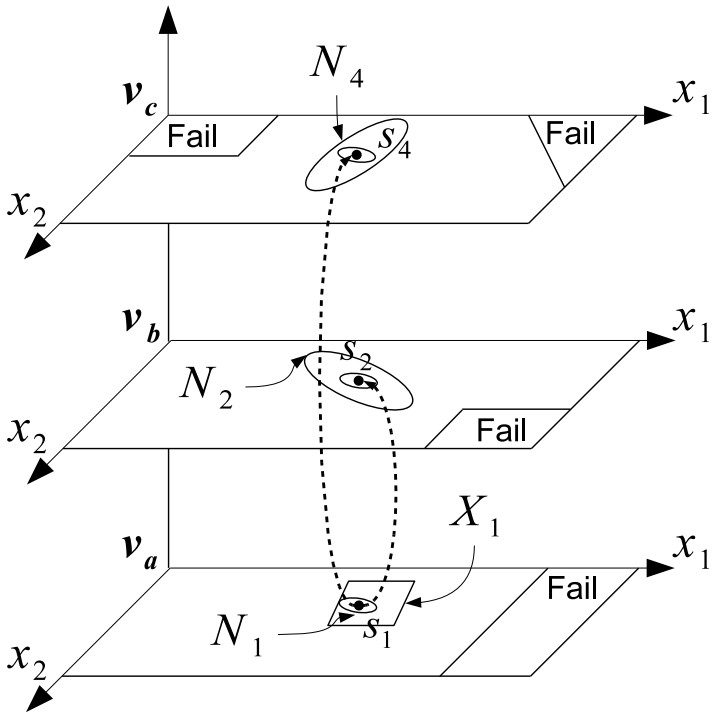



Figure 5.2: $N_1$ is safe for $\min(T_2, T_4)$ since any state in $N_1$ leads to a state within $N_2$ or $N_4$, which we assume to be safe for $T_2$ and $T_4$, respectively.

## 5.1 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 [9], but uses levels 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

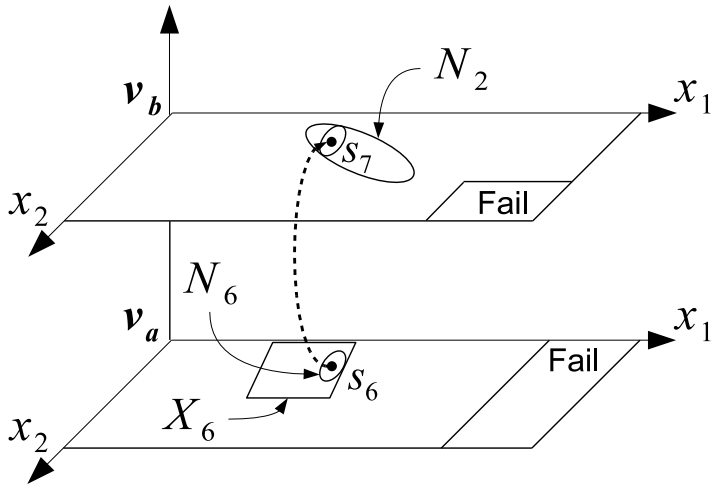



Figure 5.3: The trace containing $s_6$ and $s_7$ is merged with a previously visited trace. $N_6$ is safe for $T_2$ since any state in $N_6$ leads to a state within $N_2$, which we assume to be safe for $T_2$.

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 5.4 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 ($\text{Safe}_{Init}$ 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

```
1:  global SDCS, Fail, T;
2:  global safe_sets ← ∅; // Sets of safe plant states, initially empty.

3:  main : // Check bounded-time safety of SDCS
4:    Safe_Init ← ∅ // Set of safe initial states.
5:    foreach ((q, x) ∈ Init) // Depth-first search for each initial state.
6:      result ← explore(q, x, T, [(q, x)]);
7:      if(result = SAFE(X))
8:        Safe_Init ← Safe_Init ∪ {(q, x) | x ∈ 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 tau.
13:   if ((q, x) ∈ Fail) return UNSAFE(σ); // Check for fail states
14:   if (is_deadlock(q, x)) return DEADLOCK(σ) // Detect deadlocks
15:   if (is_livelock(q, x, σ)) return LIVELOCK(σ) // Detect livelocks

16:   if (∃ (q̂, X̂, τ̂) ∈ safe_sets : q = q̂ ∧ x ∈ X̂ ∧ τ ≤ τ̂)
17:     return SAFE(X̂); // Merge traces if within a safe set.
18:   if (q.L = L_final)
19:     result = plant_transition(q, x, τ, σ); // Plant transition
20:   else
21:     result = controller_transitions(q, x, τ, σ); // Supervisor transitions
22:   if (result = SAFE(X))
23:     safe_sets ← safe_sets ∪ {(q, X, τ)}; // Plant states safe for τ at q.
24:   return result;
```

Figure 5.4: The conservative merging verification algorithm.

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

36: **function** controller_transitions($q,\ \mathbf{x},\ \tau,\ \sigma$)
37: $\hat{Q}\ \leftarrow\ \{\hat{q} \,|\, \exists\ i :\ (q,\ \hat{q})\ \in\ \delta_i(\mathbf{x})\}$; *// Explore each successor*
38: $\mathcal{X}\ \leftarrow\ [\mathbf{x}]_q\ \cap\ \overline{\mathit{Fail}}$;
39: **foreach** ($\hat{q}\ \in\ \hat{Q}$)
40: result $\leftarrow$ explore($\hat{q},\ \mathbf{x},\ \tau,\ \sigma\ \cdot\ (\hat{q},\ \mathbf{x})$);
41: **if** (result $=$ SAFE($\hat{\mathcal{X}}$))
42: $\mathcal{X}\ =\ \mathcal{X}\ \cap\ \hat{\mathcal{X}}$;
43: **else**
44: **return** result;
45: **return** SAFE($\mathcal{X}$);

Figure 5.4: Continued – The conservative merging verification algorithm.

on line 11.

The procedure `explore` takes as arguments a state $(q, \mathbf{x})$, a time bound $\tau$, and a trace $\sigma$ which leads to $(q, \mathbf{x})$. The time bound $\tau$ 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 $\sigma$ is used to generate a counterexample if a fail state is reached (line 13), a deadlock is detected (line 14), or the current state is a livelock state (line 15).

The current state is compared with the sets of safe states that have been determined so far (lines 16-17). 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 $L_{\mathit{final}}$, then a plant transition is performed by calling the function `plant_transition` (line 19). Otherwise, the transitions of the supervisor are explored by calling the function `supervisor_transitions` (line 21). 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 $\tau$ is added to the list of safe sets (line 23).

The result of a plant transition is computed by the function `plant_transition` in Figure 5.4. Line 27 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 $\tau$ at $q$ is simply the set of plant states that are not fail states of supervisor state $q$, since $\tau < t_s$ (line 27). Otherwise, the successor state $(\hat{q}, \hat{\mathbf{x}})$ of the current state $(q, \mathbf{x})$ is computed using numerical simulation (line 28) and by setting the current control location to $L_{initial}$ (line 29). 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 30).

If the result at line 30 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 inner levels set of the bisimulation function centered around $\mathbf{x}$ that is safe for time bound $\tau$ 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 32). The set $\mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{x}, r_{max})$ is returned to the caller since it is safe for time bound $\tau$ at $q$.

The function `supervisor_transitions` in Figure 5.4 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 37). Each successor $\hat{q}$ is visited by calling the function `explore` over $(\hat{q}, \mathbf{x})$ with the same time bound $\tau$ (since supervisor transitions are instantaneous) and with

a trace that adds the new state $(\hat{q}, \mathbf{x})$ to $\sigma$ (line 40).

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 45), computed by lines 38 and 42, is

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

The set $\mathcal{X}$ satisfies the hypothesis of Theorem 5.3, namely $\mathcal{X} \cap Fail_q = \emptyset$, $\mathcal{X} \subseteq [\mathbf{x}]_q$, and, for every supervisor successor $\hat{q} \in \hat{Q}$, $X \subseteq \hat{\mathcal{X}}_{\hat{q}}$.

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

**Theorem 5.4 (Correctness)** *Consider an SDCS, a set of fail states Fail, and a time bound $T$. If the algorithm of Figure 5.4 returns $SAFE(\mathcal{X})$ then the SDCS is safe for time bound $T$, $Init \subseteq \mathcal{X}$, and all states in $\mathcal{X}$ are safe for time bound $T$. If the algorithm returns $UNSAFE(\sigma)$ then the SDCS is not safe for time bound $T$ and $\sigma$ is a trace of duration less than $T$ that ends at a state in Fail. If the algorithm returns $DEADLOCK(\sigma)$ then a deadlock state is reachable within time bound $T$. If the algorithm returns $LIVELOCK(\sigma)$ then a livelock state is reachable within time bound $T$.*

**Proof**

(Sketch) We can show, by induction on the time bound $\tau$ that, given a state $(q, \mathbf{x})$ and a path $\sigma$ to $(q, \mathbf{x})$, if `explore` returns $SAFE(\mathcal{X})$, then $\mathbf{x} \in \mathcal{X}$ and $\mathcal{X}$ is safe for $\tau$ 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 $\tau$ at $q$. Otherwise, if $q = (L_{final}, \mathbf{v})$ and $\tau \geq t_s$, a plant transition is possible: in this case, by Theorem 5.2, the computed set is safe for $\tau$ at $q$. If the location of $q$ is not equal to $L_{final}$, the set of supervisor transitions is computed and, according to Theorem 5.3, the constructed set is safe for $\tau$ at $q$.

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

If the algorithm returns $UNSAFE(\sigma)$, it must have executed line 13, in check case the trace $\sigma$ ends at state $(q, \mathbf{x})$, which belongs to *Fail*. Therefore, $\sigma$ 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.

A similar argument holds if the algorithm returns $DEADLOCK(\sigma)$ or $LIVELOCK(\sigma)$. □

**Theorem 5.5 (Termination)** *Given an SDCS with a final set of initial states, the algorithm of Figure 5.4 always terminates.*

**Proof**

(Sketch) The algorithm constructs a trace $\sigma$ that leads to the current state. Every trace generated by the algorithm is finite. This is true because the trace has only a finite number of plant transition, because the algorithm generates only traces of duration less than $T$; moreover, a trace cannot contain an infinite number of supervisor transitions. In order for that to happen,

there would an infinite suffix of $\sigma$ corresponding to only supervisor transitions. The plant state of the states in the suffix would be all the same, and, since there exists a finite number of supervisor states (*Loc* and $V$ are both finite sets), one of the states in the suffix of $\sigma$ must be repeated and the algorithm would detect a livelock and terminate. This means that the recursion tree has a finite depth. Since the plant transitions are deterministic and there exists a finite set of supervisor states, the recursion tree obtained during execution of the algorithm also 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. □

## 5.2 Merging for Affine Dynamics

In this section, 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 $f_{\mathbf{v}}(\mathbf{x}) = \mathbf{A}_{\mathbf{v}}\mathbf{x} + \mathbf{B}_{\mathbf{v}}$, a bisimulation function is given by

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

where $\mathbf{P}_{\mathbf{v}}$ satisfies the Lyapunov inequality ${\mathbf{A}_{\mathbf{v}}}^T \mathbf{P}_{\mathbf{v}} + \mathbf{P}_{\mathbf{v}} \mathbf{A}_{\mathbf{v}} \leq \mathbf{0}$ [6, 16]. The inner levels sets are given by $\mathcal{N}_{\varphi_{\mathbf{v}}}(\mathbf{x}, r) = \{\mathbf{z} \in \mathbb{R}^n \mid (\mathbf{z} - \mathbf{x})^T \mathbf{P}_{\mathbf{v}} (\mathbf{z} - \mathbf{x}) \leq r\}$,

which are ellipsoidal set [19].

**Maximum Ellipsoid Within an Ellipsoid.** In the case of affine dynamics, an operation required by the procedure given in Section 5.1 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 [4] that this is equivalent to the following:

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

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 Linear Matrix Inequality (LMI) constraints [4]. Numerical tools exists for solving such problems in polynomial time [12].

**Maximum Ellipsoid Within a Set of Linear Constraints.** The other operation required by the procedure given in Section 5.1 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 [19]

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

## 5.3 Experimental Evaluation

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 [27]. 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 [12, 26] 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 [15]. The vehicle, shown in Figure 5.5, 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 ($\theta$ 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 5.5. Equivalent force is applied by motors 2 and 4 at all times.

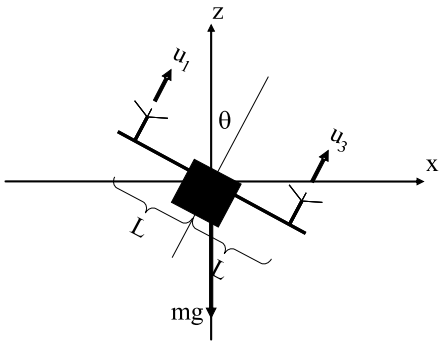



Figure 5.5: An illustration of the dynamics captured by the enhanced STARMAC model.

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 5.6 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 is probably due to the fact that the operations involving storing and lookup of ellipsoids have not been optimized.

| | Model-Checking-Guided Simulation without Merging | Model Checking with Safe Sets and Merging |
|---|---|---|
| Visited states | 43,134 | 25,493 |
| Running time | 17 sec | 251 sec |
| Memory usage | 90.2MB | 64.7MB |

Figure 5.6: 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.

# Chapter 6

# Timeline

Work on the topics in this proposal is already underway. Most of the techniques presented here have already been implemented in a prototype tool. I would like to further develop the implementation, apply it to additional examples, and extend the existing algorithms to handle more complex systems.

**Systematic Simulation.** The approach described in Chapter 3 has been implemented and has been successful in identifying an error in a realistic example. The implementation is focused on the case of task interleaving. I would like to look in more detail at models that include a controller implemented as a set of distributed processes communicating over a shared channels. This type of controllers are commonly used in industry. This effort would benefit not only the systematic simulation approach itself, but also the other approaches in this thesis. If time permits, I would like to integrate the tool with an existing code generator for Stateflow so that the

tool could be used directly on MATLAB/Simulink models without manual intervention.

**Approximate Equivalence.** Chapter 4 describes an approach for testing a system by exploring a subset of its behaviors. The behaviors that are explored depends on a notion of *approximate equivalence* between states. I plan to develop and evaluate a few different types of approximate equivalence relations in order to determine the benefits of the different notions of equivalence as well as their effectiveness.

**Conservative Merging.** The work in Chapter 5 is the most recent but also the most interesting. The current implementation demonstrates that a number of optimizations are necessary to make this approach scale to larger systems. In this thesis, I plan to explore various ways to improve the performance of the approach. One current bottleneck is the fact that ellipsoidal sets can only shrink while propagating them backward on a trace. However, based on the dynamics of the system, it is possible to compute a better estimate for the set of states that are safe. Moreover, the constraints associated with the equivalence classes of the program equivalence relation often lead to a further shrinking of the ellipsoidal sets: by taking into account the result of multiple simulation traces or by exploring additional traces it is possible to limit this shrinking. Another aspect that I would like to investigate is the application of this approach to non-linear dynamic systems for which bisimulation functions can be computed efficiently.

I propose the following timeline for the completion of the thesis:

- *November 2007 - January 2008*: Optimizations and heuristics for the conservative merging algorithm.
- *December 2007 - March 2008*: Extend the systematic simulation framework to distributed systems.
- *February 2008 - April 2008*: Application of conservative merging to non-linear dynamic systems.
- *February 2008 - April 2008*: Explore different notions of approximate equivalence.
- *May 2008 - June 2008*: Writing of the dissertation.
- *July 2008*: Defense of the thesis.

If time permits, I am interested in integrating the current implementation with an existing code generator for Stateflow and investigating the use of alternative software model checking techniques and tools within this framework.

# Bibliography

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

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

[3] 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.

[4] 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.

[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] Carmen Chicone. *Ordinary Differential Equations with Applications*. Springer-Verlag, 2006.

[7] 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.

[8] 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.

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

[10] Alexandre Donzé and Oded Maler. Systematic Simulation using Sensitivity Analysis. In *Proc. of the 10th International Workshop on Hybrid Systems: Computation and Control*, 2007.

[11] Antoine Girard and George J. Pappas. Approximation Metrics for Discrete and Continuous Systems. Technical Report MS-CIS-05-10, University of Pennsylvania, 2005.

[12] Michael Grant, Stephen Boyd, and Yinyu Ye. *CVX User's Guide*. 2007.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Alexander B. Kurzhanski and Istvan Vályi. *Ellipsoidal Calculus for Estimation and Control*. Birkhäuser, Boston, 1997.

[20] Flavio Lerda, James Kapinski, Edmund M. Clarke, and Bruce H. Krogh. Verification of Supervisory Control Software Using State Proximity and Merging. In *Submitted to the 11th International Workshop on Hybrid Systems: Computation and Control*, 2008.

[21] 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*, 2008.

[22] J. Lions. ARIANE 5 Flight 501 Failure. World Wide Web, July 1996. http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf.

[23] L. Ljung. *System identification: theory for the user*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[24] 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.

[25] Sebastian Scherer, Flavio Lerda, and Edmund Clarke. Model Checking of Robotic Control Systems. In *Proc. of the 8th International symposium on Artificial Intelligence, Robotics and Automation in Space*, 2005.

[26] Kim-Chuan Toh, Michael J. Todd, and Reha H. Tütüncü. *SDPT3 4.0*. MIT Press, 2006.

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