Control Software Model Checking Using Bisimulation Functions for Nonlinear Systems

James Kapinski[1], Alexandre Donzé[2], Flavio Lerda[2],
Hitashyam Maka[1], Silke Wagner[2], and Bruce H. Krogh[1]
[1] Dept. of Electrical and Computer Engineering, [2] Dept. of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213
[1]{jpk3|hmaka|krogh@ece.cmu.edu} [2]{adonze|flerda|silkwa@cs.cmu.edu}

***Abstract*— This paper extends a method for integrating source-code model checking with dynamic system analysis to verify properties of controllers for nonlinear dynamic systems. Source-code model checking verifies the correctness of control systems including features that are introduced by the software implementation, such as concurrency and task interleaving. Sets of reachable continuous states are computed using numerical simulation and bisimulation functions. The technique as originally proposed handles stable dynamic systems with affine state equations for which quadratic bisimulation functions can be computed easily. The extension in this paper handles nonlinear systems with polynomial state equations for which bisimulation functions can be computed in some cases using sum-of-squares (SoS) techniques. The paper presents the convex optimizations required to perform control system verification using a source-code model checker, and the method is illustrated for an example of a supervisory control system.**

I. Introduction

Verifying that a control system design satisfies given specifications requires a representation of the real-time control algorithm running on a computer interacting with the dynamic system being controlled. Usually control system designs are evaluated using numerical simulation. In simulation models, there is typically no explicit representation of the actual real-time control program. Recently, we introduced a method of verifying control systems using a source-code model checker to manage the exploration of the state space of the control software [4]. In this approach, sets of reachable continuous states are computed using numerical integration and the concept of bisimulation functions proposed by Girard and Pappas [3]. The technique described in [4] handles stable dynamic systems with affine state equations for which quadratic bisimulation functions can be computed easily. This paper presents an extension of our model checking technique to nonlinear systems with polynomial state equations for which bisimulation functions can be computed in some cases using sum-of-squares (SoS) techniques [6], [1].

This research was sponsored by the Air Force Office of Scientific Research (AFOSR) under contract no. FA9550-06-1-0312, the National Science Foundation (NSF) under grant no. CCR-0411152, the Gigascale Systems Research Center (GSRC), Semiconductor Research Corporation (SRC), the Naval Research Laboratory (NRL), and the General Motors Collaborative Research Lab at CMU. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of AFOSR, NSF, GSRC, SRC, NRL, GM, or the U.S. government.

Using a source-code model checker rather than simulation to verify properties of real-time control systems offers a number of advantages. Software verification applies to the actual code that implements the controller, rather than just a model of the control law. This assures that bugs introduced in the generation of the source code will be detected. It also makes it possible to verify aspects of the system behavior introduced by the software implementation, such as concurrency and task interleaving. A model checker efficiently manages the exploration of multiple paths of execution, whereas a simulation run represents only a single system trajectory. Model checkers identify states that have been visited previously so that every run of the system does not have to be executed.

In a manner similar to the work of [1] and [3], our approach uses the sublevel sets of bisimulation functions to verify the properties of entire sets of continuous-time trajectories based on individual numerical simulations. The extension to nonlinear dynamic systems presented in this paper is critical for verifying properties of supervisory controllers that must operate correctly beyond the range of conditions covered by linearized dynamic models. The paper makes the following contributions: (a) we formulate the optimization problems that need to be solved to perform model-checking-based verification for nonlinear dynamic systems; (b) we show how SoS techniques can be applied to solve these problems; (c) we discuss and illustrate practical aspects of achieving effective computational results in the context of our verification procedure; and (d) we illustrate the approach for an example of a supervisory control system for a nonlinear dynamic system. The concluding section discusses directions for further research on the theory of model checking for control system software and on methods for improving the performance of our model checking tool.

II. Verification of Control Software

This section briefly reviews the verification algorithm presented in [4]. We consider *sampled-data control systems* (SDCS) comprising a plant described by dynamic state equations and a controller implemented as software that periodically updates the commands to the plant. The behavior of the controller may be nondeterministic due to task interleaving.

A state s of an SDCS consists of a *control location* L, which corresponds to a location in the control software, a valuation of the controller variables $\mathbf{v}$, and a valuation of the plant variables $\mathbf{x}$. The pair $q = (L, \mathbf{v})$ is called the *controller state* and $\mathbf{x}$ is called the *plant state*. We will denote the SDCS state by either $s = (q, \mathbf{x})$ or $s = ((L, \mathbf{v}), \mathbf{x})$. The controller executes periodically at multiples of the sampling time t_s. We assume that the time required to execute the control program is negligible relative to the sampling period t_s. A *controller transition* corresponds to the execution of an atomic operation in the control program, which takes the controller to a new control state. When the controller executes at a sampling instant, several controller transitions can occur. We assume that every sequence of controller transitions starts at an *initial control location* L_{initial} and eventually terminates in a finite number of controller transitions at the *final control location* L_{final}. No controller transition is allowed from control location L_{final}. Since we assume that the control program execution time is negligible, the plant state does not change while controller transitions occur.

Plant dynamics are described by state equations of the form $\dot{\mathbf{x}} = f_{\mathbf{v}}(\mathbf{x})$, where the plant dynamics $f_{\mathbf{v}}(\cdot)$ depend on the valuation of the controller variables $\mathbf{v}$. Let $\xi_{\mathbf{v}}^{\mathbf{x}_0} : \mathbb{R} \rightarrow \mathbb{R}^n$ denote a solution to the initial value problem $\dot{\mathbf{x}}(t) = f_{\mathbf{v}}(\mathbf{x}(t)), \mathbf{x}(0) = \mathbf{x}_0$. We assume $f_{\mathbf{v}}(\cdot)$ is differentiable and that the solution $\xi_{\mathbf{v}}^{\mathbf{x}_0}(\cdot)$ to the state equation for given $\mathbf{v}$ and $\mathbf{x}_0$ always exists and is unique. A *plant transition* for given $\mathbf{v}$ and $\mathbf{x}_0$ corresponds to the evolution of the dynamical system for the sampling time t_s from $\mathbf{x}_0$ to $\xi_{\mathbf{v}}^{\mathbf{x}_0}(t_s)$. Plant transitions are allowed only from control location L_{final} (i.e., when the controller's execution has terminated). Controller variables do not change during plant transitions but the control location is reset to L_{initial} at the end of a plant transition.

A *trace* of an SDCS is a finite sequence of states $\sigma = s_0 \ldots s_K$ such that $s_k \longrightarrow s_{k+1}$ corresponds to either a *controller transition* or a *plant transition*. The *duration* of a trace is given by the number of plant transitions multiplied by the sampling time t_s. Figure 1 illustrates traces of an SDCS. The plant states correspond to the x_1 and x_2 axes. The vertical axis corresponds to the controller variables. Plant transitions are represented by continuous arrows; sequences of controller transitions from L_{initial} to L_{final} are represented by dotted arrows.

Bounded-Time Safety: Let $Fail$ denote a set of *fail states* for the SDCS. The property we want to verify is bounded-time safety: Given a time bound T (assumed to be a multiple of the sampling period t_s) and a set of initial states I, we want to prove that no trace of the SDCS that starts at an initial state $s_0 \in I$ and whose duration is less than the time bound T contains a system state that is in *Fail*.

Our approach for checking bounded-time safety of an SDCS is based on performing simulations of a subset of its traces while pruning some of the traces by merging states. Model checking merges only identical states, while our approach performs a merge also when two states are in proximity to each other, provided the pruned traces are guaranteed to be safe. The pruned traces are shown to be safe by using bisimulation functions (presented formally in the following section) to compute sets of safe plant states around the points in a trace. These sets correspond to traces that are in proximity of the visited trace and are guaranteed to be safe. When the algorithm reaches a state within a safe set, the corresponding trace is pruned.

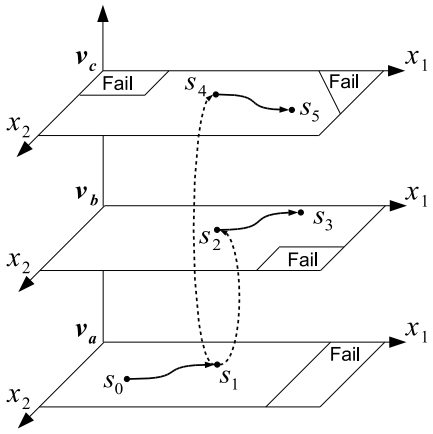


Fig. 1. An illustration of the traces of an SDCS. Solid arrows represent plant transitions. Dotted arrows represent sequences of controller transitions.

The algorithm computes *safe sets* surrounding points on the simulation traces and propagates them backwards from the end points. A set of plant states X is a *safe set* for controller state q and time bound τ if and only if for every plant state $\mathbf{x} \in X$, every trace of the SDCS starting at $(q, \mathbf{x})$ of duration less than or equal to τ does not reach a state in the $Fail$ set.

In general, given a dynamical system and two initial states that are in proximity to each other, the trajectories starting at those states may diverge. Bisimulation functions were introduced by Girard and Pappas as a way to extend the notion of bisimulation relations from discrete systems to continuous systems [2]. We use bisimulation functions to determine safe sets of plant states and then to propagate safe sets backward in time along plant transitions.

To characterize the control program behavior for sets of continuous states we define the notion of *program equivalence*. Let the set of discrete successors of a state $(q, \mathbf{x})$, denoted by $\hat{Q}(q, \mathbf{x})$, be the set of controller states $\hat{q}$ such that $(q, \mathbf{x}) \longrightarrow (\hat{q}, \mathbf{x})$ is a controller transition. Given two plant states $\mathbf{x}'$ and $\mathbf{x}''$ and a controller state q, we say that $\mathbf{x}'$ and $\mathbf{x}''$ are program equivalent at q, denoted by $\mathbf{x}' \approx_q \mathbf{x}''$, if and only if the set of discrete successors of $(q, \mathbf{x}')$ is equal to the set of discrete successors of $(q, \mathbf{x}'')$, i.e., $\hat{Q}(q, \mathbf{x}') = \hat{Q}(q, \mathbf{x}'')$. The equivalence class of $\mathbf{x}'$ with respect to $\approx_q$, denoted by $[\mathbf{x}']_{\approx_q}$, is a set of plant states that cannot be distinguished by the program (i.e., the program generates the same successors for all states in the set). The equivalence classes of the program equivalence relation are used to compute a safe set for a given state given the safe sets for the states reachable from it by performing controller

transitions.

The algorithm presented in [4] assumes that plant dynamics are provided as stable, affine state equations, for which quadratic bisimulation functions can be computed by solving Lyapunov equations. Sublevel sets of the computed bisimulation functions (defined in the next section) are used to approximate the safe sets computed by the algorithm. The technique requires several optimizations be performed on the fly. These are needed when propagating the safe sets backward along a trace. The two basic operations required are: (i) maximizing the size of a sublevel set subject to a set of constraints; and (ii) maximizing the size of a sublevel set so that it remains within a given sublevel set. Section IV addresses how these operations can be performed for a class of nonlinear systems.

III. Bisimulation Functions

To perform the algorithm described in Sec. II, a bisimulation function must be computed for each possible value of the controller variables $\mathbf{v}$. The bisimulation functions are used to produce conservative estimates of the set of reachable plant states. This section describes methods for computing bisimulation functions for nonlinear systems.

We begin with the definition of a bisimulation function for a dynamical system

$$\Sigma \; : \; [\dot{\mathbf{x}}(t) = f(\mathbf{x}(t)), \mathbf{x} \in \mathbb{R}^n] \, ,$$

whose transition relation $\rightarrow$ is given by

$\mathbf{x} \xrightarrow{t} \mathbf{x}'$ iff $\exists \xi : [0,t] \rightarrow \mathbb{R}^n$ such that
$\xi(0) = \mathbf{x}, \; \xi(t) = \mathbf{x}', \;$ and $\forall \bar{t} \in [0,t] : \dot{\xi}(\bar{t}) = f(\xi(\bar{t}))$.

We define a bisimulation function to be a differentiable function $\varphi : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$ that satisfies the following requirements [3]:

$$\varphi(\mathbf{x}_1, \mathbf{x}_2) \geq 0 \;\; \text{and} \tag{1}$$

$$\frac{\partial \varphi(\mathbf{x}_1, \mathbf{x}_2)}{\partial \mathbf{x}_1} f(\mathbf{x}_1) + \frac{\partial \varphi(\mathbf{x}_1, \mathbf{x}_2)}{\partial \mathbf{x}_2} f(\mathbf{x}_2) \leq 0 \tag{2}$$

for every $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$.

Given a bisimulation function φ for a dynamical system Σ, a state $\mathbf{x} \in \mathbb{R}^n$ of Σ, and a real value $r \geq 0$, the *sublevel set* around $\mathbf{x}$ of size r, denoted by $\mathcal{N}_\varphi(\mathbf{x}, r)$, is defined as

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

The following proposition ensures that our approximation of reachable plant states using bisimulation functions is conservative.

Proposition 1: Let φ be a bisimulation function, $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$ be arbitrary states of Σ, $t \geq 0$, $r \geq 0$, and $\mathbf{x}_i \xrightarrow{t} \mathbf{x}_i'$ for $i \in \{1, 2\}$. Then the following holds:

$$\mathbf{x}_2 \in \mathcal{N}_\varphi(\mathbf{x}_1, r) \implies \mathbf{x}_2' \in \mathcal{N}_\varphi(\mathbf{x}_1', r)$$

The proof of this proposition is a direct consequence of Corollary 1 of [3]. In the following, we present a technique for computing bisimulation functions for a class of nonlinear systems using sum-of-squares (SoS) techniques inspired from [1].

A multivariate polynomial, $p(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$, is a sum-of-squares (SoS) if there exists polynomials $s_i(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$, for $i \in \{1, \ldots, m\}$, such that

$$p(\mathbf{x}) = \sum_{j=1}^{m} s_i^2(\mathbf{x}).$$

Let $\mathbb{S}$ be the set of all SoS. Efficient techniques exist for determining that a polynomial is a SoS [6], [7], [5]. Also, optimization problems involving constraints requiring that polynomials be SoS can be posed as semi-definite programming problems and solved using efficient numerical techniques.

For problems where we require that a polynomial be positive, we instead require that the polynomial be a SoS, which is an easier problem to solve. For the computation of a bisimulation function φ for a dynamical system Σ this means that, instead of computing φ such that (1) and (2) hold, we compute φ such that

$$\varphi \in \mathbb{S} \text{ and } -D\varphi \triangleq -\left(\frac{\partial \varphi}{\partial \mathbf{x}_1} f(\mathbf{x}_1) + \frac{\partial \varphi}{\partial \mathbf{x}_2} f(\mathbf{x}_2) \right) \in \mathbb{S} \, . \tag{3}$$

Assume that the plant dynamics and the bisimulation function are polynomial. We select the degree and the form of the bisimulation function (i.e., the degree and the monomial terms that occur in the polynomial bisimulation function) and then pose a convex feasibility problem. The solution to this problem identifies the coefficients in the bisimulation function such that the bisimulation function satisfies (1) and (2)

For a given dynamic state equation, it may not be possible to find a bisimulation function that satisfies (3) for the entire state space. If we are only interested in behaviors within a subset of the state space, as suggested in [6], we can loosen the restrictions on the bisimulation functions so that (3) must hold only within a subset of the state space. The following describes how this is accomplished.

We first choose two functions g_1 and g_2 such that the set

$$\{(\mathbf{x}_1, \mathbf{x}_2) \in \mathbb{R}^n \times \mathbb{R}^n \mid g_1(\mathbf{x}_1) \geq 0 \, \wedge \, g_2(\mathbf{x}_2) \geq 0\} \tag{4}$$

represents a region of the state space for which we want to find a bisimulation function.[1] The conditions (1) and (2) thus become

$$\begin{array}{c} \forall (\mathbf{x}_1, \mathbf{x}_2) \text{ such that } g_1(\mathbf{x}_1) \geq 0 \, \wedge \, g_2(\mathbf{x}_2) \geq 0, \\ \varphi(\mathbf{x}_1, \mathbf{x}_2) \geq 0 \text{ and } D\varphi(\mathbf{x}_1, \mathbf{x}_2) \leq 0. \end{array} \tag{5}$$

Then we relax (5) into the following SoS problem formulation:

Find φ, $D\varphi$, and s_i for $1 \leq i \leq 4$ such that:

$$\varphi(\mathbf{x}_1, \mathbf{x}_2) - s_1(\mathbf{x}_1)\, g_1(\mathbf{x}_1) - s_2(\mathbf{x}_2)\, g_2(\mathbf{x}_2) \;\; \in \mathbb{S} \, , \tag{6}$$

$$-D\varphi - s_3(\mathbf{x}_1)\, g_1(\mathbf{x}_1) - s_4(\mathbf{x}_2)\, g_2(\mathbf{x}_2) \;\; \in \mathbb{S} \, , \tag{7}$$

$$s_i(\mathbf{x}) \;\; \in \mathbb{S} \quad \forall i \in \{1, \ldots, 4\} \, . \tag{8}$$

[1] Note that the technique generalizes in a straigthforward way to a set defined by an arbitrary number of functions.

It is easy to see that if (6), (7) and (8) are satisfied then (5) holds. Efficient numerical techniques can then be used to solve the constraints (6), (7), and (8) [6]. The idea of introducing the additional unknown polynomials s_i to solve this problem is a generalization of the so-called *S-procedure* [10].

The form of the bisimulation must be selected before the coefficients can be computed. The bisimulation function that is computed can be constrained to have certain desirable properties. Since we are ultimately interested in sublevel sets of the bisimulation functions, it is essential to know at least one point that is within each sublevel set. For every $\mathcal{N}_\varphi(\mathbf{x}, r)$, we call $\mathbf{x}$ the *center* of $\mathcal{N}_\varphi(\mathbf{x}, r)$. We select the form of the bisimulation function such that the center of every sublevel set of the bisimulation function is contained within the sublevel set.

Property 1: For every $\mathbf{x} \in \mathbb{R}^n$ the following holds: if a bisimulation function satisfies $\varphi(\mathbf{x}, \mathbf{x}) = 0$, then for every $r \in \mathbb{R}$, $\mathbf{x} \in \mathcal{N}_\varphi(\mathbf{x}, r)$.

We choose the following form for the bisimulation function:

$$\varphi(\mathbf{x}_1, \mathbf{x}_2) \;=\; \mathbf{Z}^T \mathbf{M} \mathbf{Z}, \tag{9}$$

where $\mathbf{Z}$ is a vector of length m with entries Z_j for $j \in \{1, \ldots, m\}$. Each entry Z_j of $\mathbf{Z}$ is given by a polynomial of the form

$$Z_j \;=\; (x_{1,k} - x_{2,k})^{h_j}$$

for $h_j \in \mathbb{N}^+$ and $k \in \{1, \ldots, n\}$, where $x_{i,k}$ denotes the k-th entry of the vector $\mathbf{x}_i$, $i = 1, 2$. In this case we have $\varphi(\mathbf{x}, \mathbf{x}) = 0$ and Property 1 holds.

To compute the bisimulation function φ we select the polynomials in $\mathbf{Z}$ and the size of the matrix $\mathbf{M}$. The entries of $\mathbf{M}$ are the decision variables.

If we also have that each Z_j is given by

$$Z_j \;=\; (x_{1,j} - x_{2,j}),$$

then the following holds: i.) $\sqrt{\varphi(\cdot, \cdot)}$ is a pseudo metric (that is, one may have $\sqrt{\varphi(\mathbf{x}_1, \mathbf{x}_2)} = 0$ for distinct values $\mathbf{x}_1 \neq \mathbf{x}_2$) and ii.) the set $\mathcal{N}_\varphi(\mathbf{x}, r)$ is ellipsoidal.

IV. Optimizing Polynomial Sublevel Sets

The verification framework that we use requires that we perform certain convex optimizations involving sublevel sets and constraints imposed by the software. In this section, we describe how to optimize the size of a polynomial sublevel set subject to constraints. The operations are used by our verification technique to compute maximally safe sublevel sets and to perform the merging operation. We provide a means by which the problems can be solved numerically using convex optimization techniques.

A. Optimizing Sublevel Sets

To compute safe sets, the sizes of a polynomial sublevel sets are maximized subject to the linear constraints from the software that characterize program equivalence sets. The following describes how this optimization problem is formulated and solved.

Let $\mathbf{z}$ be a state in $\mathbb{R}^n$ and consider the conjunction of linear constraints

$$\bigwedge_{i \in \mathcal{I}} \mathbf{l}_i^T \mathbf{x} \leq d_i,$$

for some set $\mathcal{I} = \{1, \ldots, i_{max}\}$, where each $\mathbf{l}_i \in \mathbb{R}^n$ and $d_i \in \mathbb{R}$. To maximize the size of a sublevel set of φ around $\mathbf{z}$, $\mathcal{N}_\varphi(\mathbf{z}, r)$, subject to these constraints, we solve the following optimization problem:

$$\begin{array}{ll} \text{maximize} & r \\ \text{subject to} & \forall \mathbf{x}: \; \bigwedge_{i \in \mathcal{I}} [\varphi(\mathbf{z}, \mathbf{x}) \leq r \Rightarrow \mathbf{l}_i^T \mathbf{x} \leq d_i]. \end{array} \tag{10}$$

Applying an S-procedure and introducing a new unknown variable $\lambda_i > 0$ for each i in $\mathcal{I}$, we can relax the constraint in (10) to the following:

$$\forall \mathbf{x}: \; \bigwedge_{i \in \mathcal{I}} [\varphi(\mathbf{z}, \mathbf{x}) - r - \lambda_i (\mathbf{l}_i^T \mathbf{x} - d_i) \geq 0]$$

Because φ is a polynomial, the constraints can be further relaxed to yield the following optimization problem with SoS constraints:

$$\begin{array}{ll} \text{maximize} & r \\ \text{subject to} & \bigwedge_{i \in \mathcal{I}} [\varphi(\mathbf{z}, \mathbf{x}) - r - \lambda_i (\mathbf{l}_i^T \mathbf{x} - d_i) \in \mathbb{S}] \\ & \lambda_i > 0 \text{ for each } i \in \mathcal{I}. \end{array} \tag{11}$$

Since φ is a polynomial and the only decision variables are r and λ_i, (11) is an SoS program.

To use sublevel sets of polynomial functions to perform the model checking technique described above, we must also be able to compute the maximum sized sublevel set $\mathcal{N}_{\varphi_2}(\mathbf{z}_2, r_2)$ of a polynomial function such that it is contained within the sublevel set $\mathcal{N}_{\varphi_1}(\mathbf{z}_1, r_1)$ of a second polynomial. We must solve the following optimization problem:

$$\begin{array}{ll} \text{maximize} & r_2 \\ \text{subject to} & \forall \mathbf{x}: \\ & [\varphi_2(\mathbf{z}_2, \mathbf{x}) \leq r_2 \Rightarrow \varphi_1(\mathbf{z}_1, \mathbf{x}) \leq r_1]. \end{array} \tag{12}$$

Applying the S-procedure and introducing a new unknown variable $\lambda > 0$ we can relax the constraint in (12) to the following:

$$\forall \mathbf{x}: \; [\varphi_2(\mathbf{z}_2, \mathbf{x}) - r_2 - \lambda(\varphi_1(\mathbf{z}_1, \mathbf{x}) - r_1) \geq 0]$$

Since φ_1 and φ_2 are polynomials, the constraints can be further relaxed, yielding the following optimization problem with SoS constraints:

$$\begin{array}{ll} \text{maximize} & r_2 \\ \text{subject to} & \varphi_2(\mathbf{z}_2, \mathbf{x}) - r_2 - \lambda(\varphi_1(\mathbf{z}_1, \mathbf{x}) - r_1) \in \mathbb{S} \\ & \lambda > 0. \end{array} \tag{13}$$

Since φ_1 and φ_2 are polynomials and the only decision variables are r_2 and λ, (13) is an SoS program.

V. EXAMPLE

Our verification technique is implemented using the explicit-state source-code model checker Java PathFinder [9]. We use the SoS tools built into the YALMIP optimization package with the SeDuMi convex optimization package to solve the optimization problems that arise during the verification [5], [8]. A Runge-Kutta numerical integration algorithm is used to provide the simulations of the nonlinear system over sample periods of duration t_s.

We consider a computer controlled system with two plant state variables. The plant dynamics are derived from an example in a previous paper [6]. The supervisory controller measures the state of the plant and produces an output that represents the desired system mode. In this model, the supervisor represents a high-level controller that switches modes of the lower-level controller (e.g., a PID loop with a sampling rate that is fast with respect to the sampling rate of the supervisor). The purpose of the supervisory controller is to move the system through a series of waypoints.

The system switches between three modes, beginning with Mode 1. When the controller detects that the plant state is within the set

$$GUARD_{1\rightarrow 2} = \{\mathbf{x}| -0.5 \leq x_1 \leq 0.5 \wedge -0.5 \leq x_2 \leq 0.5\},$$

it will switch to Mode 2. In Mode 2, when the controller detects that the plant state is within the set

$$GUARD_{2\rightarrow 3} = \{\mathbf{x}|1.5 \leq x_1 \leq 2.5 \wedge -1.5 \leq x_2 \leq -0.5\},$$

it will switch to Mode 3. The dynamics for each mode are as follows:

Mode 1:

$$f(\mathbf{x}) \quad = \quad \left[\begin{array}{c} -x_1 - 2x_2^2 \\ -x_2 - x_1x_2 - 2x_2^3 \end{array} \right],$$

Mode 2:

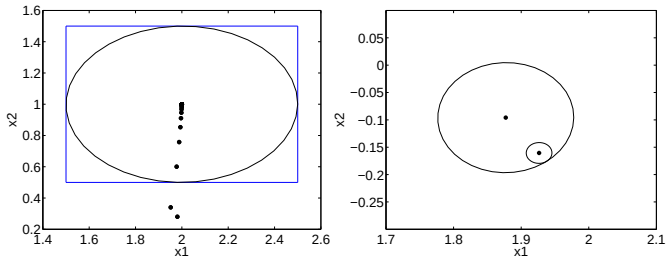
$$f(\mathbf{x}) \quad = \quad \left[\begin{array}{c} 2 - x_1 + 2(x_2+1)^2 \\ -3 - 3x_2 + x_1 + x_1x_2 - 2(1+x_2)^3 \end{array} \right],$$

Mode 3:

$$f(\mathbf{x}) \quad = \quad \left[\begin{array}{c} 4 - 2x_1 + x_1x_2 - 2x_2 - 2(x_1-2)^3 \\ 1 - x_2 + 2(x_1-2)^2 \end{array} \right].$$

The system requirement is that once the plant has entered the set $FINAL = \{\mathbf{x}|1.5 \leq x_1 \leq 2.5 \wedge 0.5 \leq x_2 \leq 1.5\}$ and the controller is in Mode 3, the plant should remain within $FINAL$ until the time bound is reached. Also, the system should satisfy the safety constraint $-2.0 \leq x_2 \leq 2.0$ for $0 \leq t \leq T_{final}$.

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 might follow slightly different traces every time a new waypoint is generated.



(a) (b)

Fig. 2. (a) First bisimulation sublevel set computed during verification; (b) Result of first merging operation during the verification.

A fourth-degree polynomial bisimulation function was computed for each of the three system modes. Each bisimulation function was selected to have the form

$$\varphi(\mathbf{x}_1, \mathbf{x}_2) \quad = \quad \mathbf{Z}^T\mathbf{M}\mathbf{Z},$$

where

$$\mathbf{Z} \quad = \quad \left[\begin{array}{c} x_1 - y_1 \\ x_2 - y_2 \\ (x_1-y_1)^2 \\ (x_1-y_1)(x_2-y_2) \\ (x_2-y_2)^2 \end{array} \right].$$

The polynomials $s_i(\mathbf{x})$ used in the constraints (6), (7), and (8) were each selected to be second degree polynomials. The functions $g_1(\mathbf{x})$ and $g_2(\mathbf{x})$ from constraints (6), (7), and (8) were chosen to define circular sets of radius 2.0 that represent regions of interest for each of the three modes.

For each of the SoS problems used to compute the bisimulation functions for Modes 1, 2, and 3, there were 235 parametric variables, 4 independent variables, 4 linear matrix inequality constraints, and 34 monomial terms. The computation times for the SoS optimizations performed to compute the bisimulation functions for each mode were 9.54, 9.43, and 10.58 seconds for Modes 1, 2, and 3, respectively.[2]

A time bound of $T_{final} = 15.0$ seconds with a sample period of $t_s = 0.5$ seconds was used for the bounded-time verification. Figures 2 and 3 illustrate the results from the computations. Figure 2-(a) shows the result of the first optimization that was performed, in which the size of a sublevel set was maximized such that it is contained within $FINAL$. Figure 2-(b) shows the result of the first merging operation. In this figure, the size of the sublevel set corresponding to a point that is being merged is maximized such that it remains within the sublevel set of the point that it is being merged with. Figure 3 shows each plant state that was visited and the sublevel set that corresponds to each visited state. Note that some of the sublevel sets are so small that they are not distinguishable from the points they are associated with. The boxes in Fig. 3 represent sets $GUARD_{1\rightarrow 2}$, $GUARD_{2\rightarrow 3}$, and $FINAL$.

[2]All computation times are for a Intel Dual Core II 2.16 GHz machine with 2GB of RAM, running Windows XP.

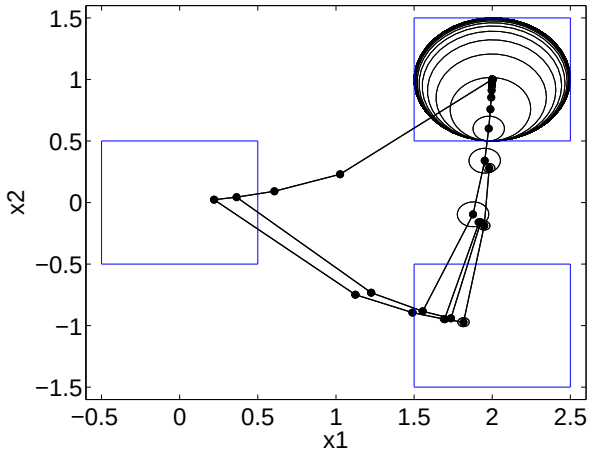


Fig. 3. Results from verification technique.

We performed the analysis both with and without state merging. The results, presented in Table I, show a significant reduction in number of visited states and memory usage. Such a reduction was obtained with only six conservative state merges: a single merge can lead to a significant reduction in visited states since every state reachable from the merged state no longer needs to be visited. The approach as implemented showed a significant overhead in terms of running time, however, which could be reduced by further optimizing the operations involving storing and lookup of sublevel sets.

TABLE I

RESULTS FROM BOUNDED-TIME VERIFICATION WITH $T_{final} = 15.0\,\text{sec}$ WITH AND WITHOUT MERGING OF SAFE STATES.

	Model-Checking-Guided Simulation without Merging	Model Checking with Safe Sets and Merging
Visited states	17,181	6,537
Running time	9.0 sec	151.0 sec
Memory usage	24.0MB	15.4MB

We encountered several challenges in computing the bisimulation functions. If a bisimulation function is not found for a selected form of $\mathbf{Z}$ in (9), there is no way to determine whether it is because no bisimulation function exists or whether one exists for some other form of $\mathbf{Z}$. A related issue is that of determining an appropriate analysis region for the S-procedure. Methods for selecting these regions such that they contain the area of interest and satisfy (6) and (7) should be investigated. Also, the behavior of the polynomial functions are sensitive to changes in the coefficient terms. For some of our experiments, truncation of coefficient terms to five decimal places caused the positive definite polynomial bisimulation functions to produce negative values. Care should be taken in manipulating the coefficients, and work should be done to develop methods for making the bisimulation function solutions more robust.

VI. DISCUSSION

This paper presents how to use bisimulation functions for nonlinear dynamic systems to aid in the verification of control software for sampled-data control systems using a source-code model checker. This extends previous work that applied only to affine, stable dynamic systems.

Although the theoretical framework presented in this paper for performing model-checking-based verification looks promising, further research is needed to make this approach valuable for a broad range of control systems. We aim to expand the type of specifications that can be verified beyond the simple safety specifications considered thus far. Another direction for research is the use of abstractions that will make it possible to handle longer time horizons.

As noted in Sec. V, there are issues to be addressed in the application of SoS tools to computation of bisimulation functions for systems with polynomial dynamics. We found that due to numerical issues, it is difficult to handle systems with more than a few state variables using current SoS tools. There are also many improvements to be made in the efficiency of the model checking implementation. Data structures and the iterations in our current implementation should be optimized for performance.

REFERENCES

[1] Antoine Girard and George J. Pappas. Approximate Bisimulations for Nonlinear Dynamical Systems. In *Proc. of the 44th Conference on Decision and Control*, 2005.

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

[3] 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.

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

[5] J. Löfberg. Yalmip : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.

[6] Pablo A. Parrilo. *Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization*. PhD thesis, California Institute of Technology, 2000.

[7] S. Prajna, A. Papachristodoulou, P. Seiler, and P. A. Parrilo. *SOSTOOLS: Sum of squares optimization toolbox for MATLAB*, 2004.

[8] J. F. Sturm. Using SeDuMi 1.02, A MATLAB Toolbox for Optimization over Symmetric Cones. *Optimization Methods and Software*, 11/12(1-4):625–653, 1999.

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

[10] Vladimir A. Yakubovich, G. A. Leonov, and A. Kh. Gelig. *Stability of Stationary Sets in Control Systems With Discontinuous Nonlinearities (Series on Stability, Vibration and Control of Systems, Series a, Vol. 14)*. World Scientific Publishing Company, 2004.