Scalable Verification of Linear Controller Software

Junkil Park[1], Miroslav Pajic[2], Insup Lee[1], and Oleg Sokolsky[1]

[1] Department of Computer and Information Science, University of Pennsylvania
{park11, lee, sokolsky}@cis.upenn.edu
[2] Department of Electrical and Computer Engineering, Duke University
miroslav.pajic@duke.edu

Abstract. We consider the problem of verifying software implementations of linear time-invariant controllers against mathematical specifications. Given a controller specification, multiple correct implementations may exist, each of which uses a different representation of controller state (e.g., due to optimizations in a third-party code generator). To accommodate this variation, we first extract a controller's mathematical model from the implementation via symbolic execution, and then check input-output equivalence between the extracted model and the specification by similarity checking. We show how to automatically verify the correctness of C code controller implementation using the combination of techniques such as symbolic execution, satisfiability solving and convex optimization. Through evaluation using randomly generated controller specifications of realistic size, we demonstrate that the scalability of this approach has significantly improved compared to our own earlier work based on the invariant checking method.

1 Introduction

Control systems are at the core of many safety- and life-critical embedded applications. Ensuring the correctness of these control system implementations is an important practical problem. Modern techniques for the development of control systems are model driven. Control design is performed using a mathematical model of the system, where both the controller and the plant are represented as sets of equations, using well established tools, such as Simulink and Stateflow.

Verification of the control system and evaluation of the quality of control is typically performed at the modeling level [3]. Once the control engineer is satisfied with the design, a software implementation of the controller is produced from the model using a generator such as Simulink Coder. To ensure that the generated implementation of the controller is correct with respect to its model, we ideally would like to have verified code generators that would guarantee that any generated controller correctly implements its model. In practice, however, code generators for control software are complex tools that are not easily amenable to formal verification, and are typically offered as black boxes. Subtle bugs have been found in earlier versions of commercially available code generators [22].

In the absence of verified code generators, it is desirable to verify instances of generated code against their models. In this paper, we consider an approach to perform such instance verification. Our approach is based on extracting a model from the controller code and establishing equivalence between the original and the extracted models. We limit our attention to linear time-invariant (LTI) controllers, since these are the most commonly used controllers in control systems. In such controllers, relations between the values of inputs and state variables, and between state variables and outputs, are linear functions of input and state variables with constant (i.e., time-invariant) coefficients.

Our technical approach relies on symbolic execution of the generated code. Symbolic expressions for state and output variables of the control function are used to reconstruct the model of the controller. The reconstructed model is then checked for input-output equivalence between the original and reconstructed model, using the well-known necessary and sufficient condition for the equivalence of two minimal LTI models. Verification is performed using real arithmetic. We account for some numerical errors by allowing for a bounded discrepancy between the models. We compare two approaches for checking the equivalence; one reduces the equivalence problem to an SMT problem, while the other uses a convex optimization formulation. We compare equivalence checking to an alternative verification approach introduced in [23], which converts the original LTI model into input-output based code annotations for verification at the code level.

The paper is organized as follows. Section 2 provides necessary background on LTI systems. Section 3 introduces the approach based on code annotations. Section 4 presents model extraction from code, followed by the equivalence checking in Section 5. Section 6 evaluates the performance of the approaches. In Sections 7 and 8, we provide a brief overview of related work and conclude the paper.

2 Preliminaries

In this section, we present preliminaries on linear controllers and the structure of linear controller implementations (e.g., step function generated by Embedded Coder). We also describe a couple of motivating examples and the notations used in this paper.

The role of feedback controllers is to ensure the desired behavior of the closed-loop systems by computing inputs to the plants based on previously measured plant outputs. We consider linear LTI controllers and assume that the specifications (i.e., models) of the controllers are given in the standard *state-space representation* form

$$\begin{aligned} \mathbf{z}_{k+1} &= \mathbf{A}\mathbf{z}_k + \mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{z}_k + \mathbf{D}\mathbf{u}_k. \end{aligned} \tag{1}$$

where $\mathbf{u}_k \in \mathbb{R}^p$ denotes the input vector to the controller at time k, $\mathbf{y}_k \in \mathbb{R}^m$ denotes the output vector of the controller at time k, $\mathbf{z}_k \in \mathbb{R}^n$ denotes the state vector of the controller. In addition, the size of the controller state n is

referred to as the size of the controller and we use a common assumption that the specified controller has minimal realization [26]; this implies that n is also the degree of the controller (i.e., the degree of the denominator of its characteristic polynomial). Note that the matrices $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\mathbf{B} \in \mathbb{R}^{n\times p}$, $\mathbf{C} \in \mathbb{R}^{m\times n}$ and $\mathbf{D} \in \mathbb{R}^{m\times p}$ together with the initial controller state $\mathbf{z}_0$ completely specify an LTI controller. Thus, we will let $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{z}_0)$ denote an LTI controller, or simply write $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ when the initial controller state $\mathbf{z}_0$ is zero.

The model of LTI controllers can be implemented in software as a function that takes as input the current state of the controller and a set of input sensor values, and computes control output (i.e., inputs applied to the plant) and the new state of the controller. We refer to this function as the *step function*. The step function is called by the control software periodically, or whenever new sensor measurements arrive. We assume that data is exchanged with the step function through global variables.[3] In other words, the input, output and state variables are declared in the global scope, and the step function reads both input and state variables, and updates both output and state variables as the effect of its execution. However, we note that this assumption does not critically limit our approach because it can be easily extended to support a different code interface for the step function.

2.1 Motivating Examples

We start by introducing two motivating examples that illustrate limitations of the straightforward verification based on the mathematical model from (1). This is caused by the fact that controller code might be generated by a code generator whose optimizations may potentially violate the model, while still guaranteeing the desired control functionality.

A Scalar Linear Integrator. We begin with an example from [23], where the controller should compute a scalar control input u_k as a scaled sum of all previous measurements $y_i \in \mathbb{R}, i = 0, 1, ..., k-1$ – i.e.,

$$u_k = \sum_{i=0}^{k-1} \alpha y_i, k > 1, \quad \text{and}, \quad u_0 = 0. \tag{2}$$

If the Simulink Integrator block with Forward Euler integration is used to implement this controller, the controller will be in the form of (1) as $\Sigma(1, \alpha, 1, 0)$, – i.e., $z_{k+1} = z_k + \alpha y_k, u_k = z_k$. Note that another realization of this controller could be $\hat{\Sigma}(1, 1, \alpha, 0)$ – i.e., $z_{k+1} = z_k + y_k, u_k = \alpha z_k$, resulting in a lower computational error due to finite precision computations [10]. Thus, for controller specification (2) two different controller implementations could be produced by different code generation tools, with the same input-output behavior while maintaining scaled and unscaled sums, respectively, of the previous values for y_k.

[3] This convention is used by Embedded Coder, a code generation toolbox for Matlab/Simulink.

Multiple-Input-Multiple-Output Controllers. The second example we will consider is a Multiple-Input-Multiple-Output (MIMO) controller, maintaining four states with two inputs and two outputs

$$\mathbf{z}_{k+1} = \underbrace{\begin{bmatrix} -0.500311 & 0.16751 & 0.028029 & -0.395599 & -0.652079 \\ 0.850942 & 0.181639 & -0.29276 & 0.481277 & 0.638183 \\ -0.458583 & -0.002389 & -0.154281 & -0.578708 & -0.769495 \\ 1.01855 & 0.638926 & -0.668256 & -0.258506 & 0.119959 \\ 0.100383 & -0.432501 & 0.122727 & 0.82634 & 0.892296 \end{bmatrix}}_{\mathbf{A}} \mathbf{z}_k + \\ + \underbrace{\begin{bmatrix} 1.1149 & 0.164423 \\ -1.56592 & 0.634384 \\ 1.04856 & -0.196914 \\ 1.96066 & 3.11571 \\ -3.02046 & -1.96087 \end{bmatrix}}_{\mathbf{B}} \mathbf{u}_k \tag{3}$$

$$\mathbf{y}_k = \underbrace{\begin{bmatrix} 0.283441 & 0.032612 & -0.75658 & 0.085468 & 0.161088 \\ -0.528786 & 0.050734 & -0.681773 & -0.432334 & -1.17988 \end{bmatrix}}_{\mathbf{C}} \mathbf{z}_k \tag{4}$$

This controller requires $25 + 10 = 35$ multiplications to update the state $\mathbf{z}$ in each step function. Similarly, in the general case, for any controller with the model in (1), $n^2 + np = n(n + p)$ multiplications are needed to update the controller's state. On the other hand, consider the controller below that requires only $5 + 10 = 15$ multiplications to update its state

$$\hat{\mathbf{z}}_{k+1} = \underbrace{\begin{bmatrix} 0.87224 & 0 & 0 & 0 & 0 \\ 0 & 0.366378 & 0 & 0 & 0 \\ 0 & 0 & -0.540795 & 0 & 0 \\ 0 & 0 & 0 & -0.332664 & 0 \\ 0 & 0 & 0 & 0 & -0.204322 \end{bmatrix}}_{\hat{\mathbf{A}}} \hat{\mathbf{z}}_k + \\ + \underbrace{\begin{bmatrix} 0.822174 & -0.438008 \\ -0.278536 & -0.824313 \\ 0.874484 & 0.858857 \\ -0.117628 & -0.506362 \\ -0.955459 & -0.622498 \end{bmatrix}}_{\hat{\mathbf{B}}} \mathbf{u}_k, \tag{5}$$

$$\mathbf{y}_k = \underbrace{\begin{bmatrix} -0.793176 & 0.154365 & -0.377883 & -0.360608 & -0.142123 \\ 0.503767 & -0.573538 & 0.170245 & -0.583312 & -0.56603 \end{bmatrix}}_{\hat{\mathbf{C}}} \hat{\mathbf{z}}_k \tag{6}$$

In general, when a matrix $\mathbf{A}$ in (1) is diagonal, only $n + np = n(p + 1)$ multiplications are performed to update $\mathbf{z}_k$ in each `step` function.

In this example, the controllers Σ and $\hat{\Sigma}$ are *similar*,[4] meaning that if the same inputs $\mathbf{y}_k$ are delivered to both controllers, the outputs of the controllers

[4] We formally define the similarity transform in Section 5.

will be identical for all k, although the states maintained by the controllers will most likely be different. As a result, although it does not obey the state evolution of the initial controller Σ, the 'diagonalized' controller $\hat{\Sigma}$ provides the same control functionality as Σ at a significantly reduced computational cost – making it more suitable for embedded applications.

2.2 Problem Statements

The introduced examples illustrate that code generation tools for embedded systems could produce more efficient code that deviates from the initial controller model as specified in (1), while being functionally correct from the input-output perspective. Consequently, in this work we will focus on verification methods that facilitate reasoning about the correctness of linear controllers without relying on the state-space representation of the controller. We will compare our approach with a verification approach we introduced in [23] which, to enable verification at the code level, converts the original LTI model into input-output code annotations based on the controllers' transfer functions. Thus, we start by providing an overview of the code annotation method for LTI controllers introduced in [23].

3 Overview of Invariant-based Approach

In [23], we introduced an approach for verification of LTI controllers using the controllers' transfer functions to provide input-output based invariants for a controller defined as $\Sigma = (\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$. The controller's transfer function $\mathbf{G}(z)$, defined as $\mathbf{G}(z) = \frac{\mathbf{Y}(z)}{\mathbf{U}(z)} = \mathbf{C}(z\mathbf{I}_n - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D}$, where $\mathbf{U}(z)$ and $\mathbf{Y}(z)$ denote the z-transforms of the signals $\mathbf{u}_k$ and $\mathbf{y}_k$ respectively, is a convenient way to capture the dependency between the controller's input and output signals. In general, $\mathbf{G}(z)$ is an $m \times p$ matrix with each element $\mathbf{G}_{i,j}(z)$ being a rational function of the complex variable z. To simplify the notation in this summary, we consider Single-Input-Single-Output (SISO) controllers, meaning that the transfer function $G(z)$ takes the form

$$G(z) = \frac{\beta_0 + \beta_1 z^{-1} + \cdots + \beta_n z^{-n}}{1 + \alpha_1 z^{-1} + \cdots + \alpha_n z^{-n}}, \tag{7}$$

where n is the size of the initial controller model (referred also as the degree of the transfer function). This allows us to specify the dependency between the controller's input and output signals as the following difference equation [26]

$$y_k = \sum_{i=0}^{n} \beta_i u_{k-i} - \sum_{i=1}^{n} \alpha_i y_{k-i}, \tag{8}$$

with $y_k = 0, k < 0$, because $\mathbf{z}_0 = 0$ and $u_k = 0$, for $k < 0$. Thus, for any controller Σ a linear invariant of the form in (8) can be used to specify the relationship between controller inputs and outputs, which is invariant to any similarity transformations [26].

4 Model Extraction from Linear Controller Implementation

In order to verify a linear controller implementation against its specification, we first extract an LTI model from the implementation (i.e., step function), and then compare it to the specification (i.e., the initial model). To obtain an LTI model from the step function, it is first necessary to identify the computation of the step function based on the program semantics. By the computation of a program, we mean how the execution of the program affects the global state.[5] This is also known as the *big-step transition relation* of a program, which is the relation between states before and after the execution of the program. In the next subsection, we explain how to identify the big-step transition relation of the step function via symbolic execution.

4.1 Symbolic Execution of Step Function

According to the symbolic execution semantics [18, 7, 6], we symbolically execute the step function with symbolic inputs and symbolic controller state. When the execution is finished, we examine the effect of the step function on the global state where output and new controller state are produced as symbolic formulas.

Model extraction via symbolic execution may not be applicable to any arbitrary program (e.g., non-terminating program, file/network IO program). However, we argue that it is feasible when focusing on the linear controller implementations which are self-contained (i.e., no dependencies on external functions) and have simple control flows (e.g., for the sake of deterministic real-time behaviors). During symbolic execution, we check if each step of the execution satisfies certain rules (i.e., restrictions), otherwise it is rejected. The rules are as follows: first of all, the conditions of conditional branches should be always evaluated to concrete boolean values. We argue that the step functions of linear controllers are unlikely necessary to branch over symbolic values such as symbolic inputs or symbolic controller states. Moreover, in many cases, the upper bound of the loops of step functions are statically fixed based on the size of the controllers, so the loop condition can be evaluated to concrete values as well. This rule results in yielding the finite and deterministic symbolic execution path of the step function. The second rule is that it is not allowed to use symbolic arguments when calling the standard mathematical functions (e.g., `sin`, `cos`, `log`, `exp`) because the use of such non-linear functions may result in non-linear input-output relation of the step function. Moreover, it is also not allowed to call external libraries (e.g., file/network IO APIs, functions without definitions provided). This rule restricts the step function to be self-contained and to avoid using non-linear mathematical functions. Lastly, dereferencing a symbolic memory address is not allowed because the non-deterministic behavior of memory access is undesirable for controller implementations and may result in unintended information flow.

[5] Note that we assume that data is exchanged with the step function via global variables.

As the result of the symbolic execution of the step function, the global variables are updated with symbolic formulas. By collecting the updated variables and their new values (i.e., symbolic formulas), the big-step transition relation of the step function can be represented as a system of equations; each equation is in the following form

$$v^{(new)} = f(v_1, v_2, \ldots, v_t)$$

where t is the number of variables used in the symbolic formula f, v, v_i are the global variables, $v^{(new)}$ denotes that the variable v is updated with the symbolic formula on the right-hand side of the equation, the variable without the superscript "(new)" denotes the initial symbolic value of the variable (i.e., from the initial state before symbolic execution of the step function). We call this equation *transition equation*.

For example, we consider symbolic execution for the step function in [24], obtained from the model (5), (6); we illustrate the transition equations of the step function as follows, replacing the original variable names with new shortened names for presentation purpose only, such as `x` for `LTIS_DW.Internal_DSTATE`, `u` for `LTIS_U.u`, and `y` for `LTIS_Y.y`:

$$\begin{aligned}
\texttt{x[0]}^{(new)} &= ((0.87224 \cdot \texttt{x[0]}) + ((0.822174 \cdot \texttt{u[0]}) + (-0.438008 \cdot \texttt{u[1]}))) \\
\texttt{x[1]}^{(new)} &= ((0.366377 \cdot \texttt{x[1]}) + ((-0.278536 \cdot \texttt{u[0]}) + (-0.824312 \cdot \texttt{u[1]}))) \\
\texttt{x[2]}^{(new)} &= ((-0.540795 \cdot \texttt{x[2]}) + ((0.874484 \cdot \texttt{u[0]}) + (0.858857 \cdot \texttt{u[1]}))) \\
\texttt{x[3]}^{(new)} &= ((-0.332664 \cdot \texttt{x[3]}) + ((-0.117628 \cdot \texttt{u[0]}) + (-0.506362 \cdot \texttt{u[1]}))) \\
\texttt{x[4]}^{(new)} &= ((-0.204322 \cdot \texttt{x[4]}) + ((-0.955459 \cdot \texttt{u[0]}) + (-0.622498 \cdot \texttt{u[1]}))) \\
\texttt{y[0]}^{(new)} &= (((((-0.793176 \cdot \texttt{x[0]}) + (0.154365 \cdot \texttt{x[1]})) + (-0.377883 \cdot \texttt{x[2]})) \\
&\quad +(-0.360608 \cdot \texttt{x[3]})) + (-0.142123 \cdot \texttt{x[4]})) \\
\texttt{y[1]}^{(new)} &= (((((0.503767 \cdot \texttt{x[0]}) + (-0.573538 * \cdot \texttt{x[1]})) + (0.170245 \cdot \texttt{x[2]})) \\
&\quad +(-0.583312 \cdot \texttt{x[3]})) + (-0.56603 \cdot \texttt{x[4]})).
\end{aligned} \tag{9}$$

4.2 Linear Time-Invariant System Model Extraction

To extract an LTI model from the obtained transition equations, we first determine which variables are used to store the controller state. To do this, we examine the data flow among the variables which appear in the equations. Let V_{used} be the set of used variables which appears on the right-hand side of the transition equations. Let $V_{updated}$ be the set of updated variables which appears on the left-hand side of the transition equations. As the interface of the step function, we assume that the sets of input and output variables are given, which are denoted by V_{input} and V_{output}, respectively. We define the set of state variables V_{state} as

$$V_{state} = (V_{updated} \setminus V_{output}) \cup (V_{used} \setminus V_{input}).$$

For example, from the transition equations (9), `x[0]`, `x[1]`, `x[2]`, `x[3]` and `x[4]` are identified as controller state variables as given the input variables `u[0]` and `u[1]`, and the output variables `y[0]` and `y[1]`.

The next step is to convert the transition equations into a canonical form. We fully expand the expressions on the right-hand side of the transition equations using the distributive law. The resulting expressions are represented in the

form of the sum of products without containing any parentheses. We check if the expressions equations are linear (i.e., each product term should be the multiplication of a constant and a single variable), and otherwise, it is rejected. Finally, each transition equation is represented as the following canonical form

$$v^{(new)} = c_1 v_1 + c_2 v_2 + \cdots + c_t v_t$$

where t is the number of product terms, $v \in V_{updated}$ is the updated variable, $v_i \in V_{used}$ are the used variables, and $c_i \in \mathbb{R}$ are the coefficients. When converting the transition equations into canonical form, we regard floating-point arithmetic expressions as real arithmetic expressions. The analysis of the discrepancy between them is left for future work. Instead, in the next section, the discrepancy issue between two LTI models due to numerical errors of floating-point arithmetic is addressed as the first step toward the full treatment of the problem.

Since the transition equations in canonical form are a system of linear equations, we finally rewrite the transition equations as matrix equations. In order to do this, we first define the input variable vector $\mathbf{u} = vec(V_{input})$, the output variable vector $\mathbf{y} = vec(V_{output})$ and the state variable vector $\mathbf{x} = vec(V_{state})$ where $vec(V)$ denotes the vectorization of the set V (e.g., $vec(\{v_1, v_2, v_3\}) = [v_1, v_2, v_3]^{\mathrm{T}}$). This allows for rewriting each transition equation in terms of the state variable vector $\mathbf{x}$ and the input variable vector $\mathbf{u}$ as

$$v^{(new)} = [c_1, c_2, \ldots, c_n]\mathbf{x} + [d_1, d_2, \ldots, d_p]\mathbf{u}$$

where n is the length of the state variable vector, p is the length of the input variable vector and $c_i, d_i \in \mathbb{R}$ are constants. Finally, we rewrite the transition equations as two matrix equations as follows

$$\mathbf{x}^{(new)} = \hat{\mathbf{A}}\mathbf{x} + \hat{\mathbf{B}}\mathbf{u}$$
$$\mathbf{y}^{(new)} = \hat{\mathbf{C}}\mathbf{x} + \hat{\mathbf{D}}\mathbf{u}$$

where $\hat{\mathbf{A}} \in \mathbb{R}^{n \times n}$, $\hat{\mathbf{B}} \in \mathbb{R}^{n \times p}$, $\hat{\mathbf{C}} \in \mathbb{R}^{m \times n}$, $\hat{\mathbf{D}} \in \mathbb{R}^{m \times p}$, and for any vector $\mathbf{v} = [v_1, \ldots v_t]^{\mathrm{T}}$, we define $\mathbf{v}^{(new)} = [v_1^{(new)}, \ldots, v_t^{(new)}]^{\mathrm{T}}$.

For example, consider the transition equation about $\mathtt{y[0]}^{(new)}$ in (9), which is represented in canonical form, and then rewritten as a vector equation (i.e., equation in terms of the state and the input variable vectors) as follows

$$\begin{aligned}
\mathtt{y[0]}^{(new)} &= (((((-0.793176 \cdot \mathtt{x[0]}) + (0.154365 \cdot \mathtt{x[1]})) + (-0.377883 \cdot \mathtt{x[2]})) \\
&\quad + (-0.360608 \cdot \mathtt{x[3]})) + (-0.142123 \cdot \mathtt{x[4]})) \\
&= -0.793176 \cdot \mathtt{x[0]} + 0.154365 \cdot \mathtt{x[1]} + -0.377883 \cdot \mathtt{x[2]} \\
&\quad + -0.360608 \cdot \mathtt{x[3]} + -0.142123 \cdot \mathtt{x[4]} \\
&= [-0.793176, 0.154365, -0.377883, -0.360608, -0.142123] \cdot \mathbf{x} \quad + [0, 0] \cdot \mathbf{u}
\end{aligned}$$

where $\mathbf{x} = [\mathtt{x[0]}, \mathtt{x[1]}, \mathtt{x[2]}, \mathtt{x[3]}, \mathtt{x[4]}]^{\mathrm{T}}$, and $\mathbf{u} = [\mathtt{u[0]}, \mathtt{u[1]}]^{\mathrm{T}}$. Converting each transition equation (9) into the corresponding vector equation, we finally reconstruct the LTI model (i.e., same as (5) (6)) from the step function of [24].

Remark 1. In general, the size of the extracted model $\hat{\mathbf{\Sigma}}$ may not be equal to the size of the initial controller model $\mathbf{\Sigma}$ from (1) (i.e., n). As we assume that $\mathbf{\Sigma}$ is minimal, if the obtained model has the size less than n it would clearly have to violate input-output (IO) requirements of the controller. However, if the size of $\hat{\mathbf{\Sigma}}$ is larger than n, we consider a controllable and observable subsystem computed via Kalman decomposition [26] from the extracted model, as the $\hat{\mathbf{\Sigma}}(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \hat{\mathbf{D}})$ model extracted from the code. Note that $\hat{\mathbf{\Sigma}}$ is minimal in this case, and thus its size has to be equal to n to provide IO conformance with the initial model.

5 Input-Output Equivalence Checking between Linear Controller Models

In order to verify a linear controller implementation against an LTI specification, in the previous section we described how to extract an LTI model from the implementation. This section introduces a method to check input-output (IO) equivalence between two linear controller models: (1) the original LTI specification and (2) the LTI model extracted from the implementation.

To check the IO equivalence between two LTI models, we exploit the fact that two minimal LTI models with the same size are IO equivalent if and only if they are *similar* to each other. Two LTI models $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ and $\hat{\mathbf{\Sigma}}(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \hat{\mathbf{D}})$ are said to be *similar* if there exists a non-singular matrix $\mathbf{T}$ such that

$$\hat{\mathbf{A}} = \mathbf{T}\mathbf{A}\mathbf{T}^{-1}, \qquad \hat{\mathbf{B}} = \mathbf{T}\mathbf{B}, \qquad \hat{\mathbf{C}} = \mathbf{C}\mathbf{T}^{-1}, \qquad \text{and} \qquad \hat{\mathbf{D}} = \mathbf{D} \tag{10}$$

where $\mathbf{T}$ is referred to as the *similarity transformation matrix* [26]. Thus, given two minimal LTI models, the problem of equivalence checking between the models is reduced to the problem of finding a similarity transformation matrix for the models. The rest of this section explains how to formulate this problem as a satisfiability problem and a convex optimization problem.

5.1 Satisfiability Problem Formulation

We start by describing an approach to formulate the problem of finding similarity transformation matrices as the satisfiability problem instance when two LTI models $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ and $\hat{\mathbf{\Sigma}}(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \hat{\mathbf{D}})$ are given. Since existing SMT solvers hardly support matrices and linear algebra operations, we encode the similarity transformation matrix $\mathbf{T}$ as a set of scalar variables $\{T_{i,j} \mid 1 \leq i, j \leq n\}$ where $T_{i,j}$ is the variable to represent the element in the i-th row and j-th column of the matrix $\mathbf{T}$. The following constraints rephrase the equations of (10) in an element-wise manner

$$\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq n}\left(\sum_{1\leq k\leq n}\hat{A}_{i,k}T_{k,j} = \sum_{1\leq k\leq n}T_{i,k}A_{k,j}\right) \wedge \bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq n}\left(\hat{B}_{i,j} = \sum_{1\leq k\leq n}T_{i,k}B_{k,j}\right)$$
$$\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq n}\left(\sum_{1\leq k\leq n}\hat{C}_{i,k}T_{k,j} = C_{i,j}\right) \wedge \bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq n}\hat{D}_{i,j} = D_{i,j} \tag{11}$$

It is important to highlight that although a similarity transform always results in an IO equivalent new controller, due to finite-precision computation of the code generator performing controller optimization, it is expected that the produced controller will slightly differ from a controller that is similar to the initial controller. Consequently, there is a need to extend our input-output invariants for the case with imprecise specification of the similarity transform. To achieve this, given error bound ϵ, the following constraints extends (11) to tolerate errors up to error bound ϵ

$$\begin{gathered}
\bigwedge_{1\le i\le n}\bigwedge_{1\le j\le n} -\epsilon \le \left(\sum_{1\le k\le n} \hat{A}_{i,k}T_{k,j}\right) - \left(\sum_{1\le k\le n} T_{i,k}A_{k,j}\right) \le \epsilon \\
\bigwedge_{1\le i\le n}\bigwedge_{1\le j\le n} -\epsilon \le \hat{B}_{i,j} - \left(\sum_{1\le k\le n} T_{i,k}B_{k,j}\right) \le \epsilon \\
\bigwedge_{1\le i\le n}\bigwedge_{1\le j\le n} -\epsilon \le \left(\sum_{1\le k\le n} \hat{C}_{i,k}T_{k,j}\right) - C_{i,j} \le \epsilon \\
\bigwedge_{1\le i\le n}\bigwedge_{1\le j\le n} -\epsilon \le \hat{D}_{i,j} - D_{i,j} \le \epsilon
\end{gathered} \tag{12}$$

For example, suppose that the original LTI model $\mathbf{\Sigma}(\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D})$ from (3)(4), the reconstructed model from the implementation $\hat{\mathbf{\Sigma}}(\hat{\mathbf{A}},\hat{\mathbf{B}},\hat{\mathbf{C}},\hat{\mathbf{D}})$ from (5)(6) and the error bound $\epsilon = 10^{-6}$ are given. Having the problem instance formulated as (12), the similarity transformation matrix $\mathbf{T}$ for those models can be found using an SMT solver which supports the quantifier-free linear real arithmetic, QF_LRA for short. Due to the lack of space, only the first row of $\mathbf{T}$ is shown here

$$\begin{aligned}
T_{1,1} &= -\frac{4456819079658364698078421593 38}{8186673753052826438040304655 63} && (\approx -0.544399156750667)\\
T_{1,2} &= -\frac{1354420228830319211286205094 82}{8186673753052826438040304655 63} && (\approx -0.165442059801384)\\
T_{1,3} &= \frac{1981727763748314492512116556 28}{8186673753052826438040304655 63} && (\approx 0.242067461044165)\\
T_{1,4} &= -\frac{3512560505509989191211978953 100}{8186673753052826438040304655 63} && (\approx -0.429058064513855)\\
T_{1,5} &= -\frac{4763453450406346969899704205 90}{8186673753052826438040304655 63} && (\approx -0.581854284748456)
\end{aligned}$$

Since, for the theory of real numbers, SMT solvers use the arbitrary-precision arithmetic when calculating answers, each element of $\mathbf{T}$ is given as a fractional number of numerous digits. For instance, although it is not displayed here, $T_{5,4}$ in this example is a fraction whose numerator and denominator are numbers with more than one hundred digits. Thus, due to the infinite precision arithmetic used by SMT solvers, the scalability of the SMT formulation-based approach is questionable. This illustrates the need for a more efficient approach for similarity checking, and in the next subsection we will present a convex optimization-based approach as an alternative method.

5.2 Convex Optimization Problem Formulation

The idea behind a convex optimization based approach is to use convex optimization to minimize the difference between the initial model and the model obtained via a similarity transformation from the model extracted from the code. Specifically, we formulate the equivalence checking for imprecise specifications as a convex optimization problem defined as

$$\begin{array}{ll} \text{variables} & e \in \mathbb{R}, \mathbf{T} \in \mathbb{R}^{n \times n} \\ \text{minimize} & e \\ \text{subject to} & \epsilon \leq e, \\ & \left\| \hat{\mathbf{A}}\mathbf{T} - \mathbf{T}\mathbf{A} \right\|_\infty \leq e, \ \left\| \hat{\mathbf{B}} - \mathbf{T}\mathbf{B} \right\|_\infty \leq e, \\ & \left\| \hat{\mathbf{C}}\mathbf{T} - \mathbf{C} \right\|_\infty \leq e, \ \left\| \hat{\mathbf{D}} - \mathbf{D} \right\|_\infty \leq e \end{array} \tag{13}$$

For example, given two LTI models $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ from (3)(4) and $\hat{\mathbf{\Sigma}}(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \hat{\mathbf{D}})$ from (5)(6) and the error bound $\epsilon = 10^{-6}$, by (13), the similarity transformation matrix $\mathbf{T}$ can be found using the convex optimization solver CVX as follows

$$\mathbf{T} = \begin{bmatrix} -0.5443990427 & -0.1654425774 & 0.2420672805 & -0.4290576934 & -0.5818538874 \\ -0.4440654044 & -0.7588435418 & 0.1765807738 & 0.2799578419 & 0.5647456751 \\ -0.588433439 & -0.2004321431 & 0.6773771193 & 0.4815317446 & 0.1449186163 \\ 0.9314576739 & -0.0459172638 & 0.6095691172 & 0.3808322795 & 0.8653864392 \\ -0.2372386619 & 0.5190687755 & 0.8165534522 & -0.1493619803 & 0.1461696487 \end{bmatrix}$$

In addition, the original similarity transformation matrix $\mathbf{T}_{ori}$ used in the actual transformation from $\mathbf{\Sigma}$ to $\hat{\mathbf{\Sigma}}$ is

$$\mathbf{T}_{ori} = \begin{bmatrix} -0.5443991568 & -0.1654420598 & 0.242067461 & -0.4290580645 & -0.5818542847 \\ -0.4440652236 & -0.7588431653 & 0.1765807449 & 0.279957637 & 0.564745456 \\ -0.5884339121 & -0.2004321022 & 0.677376781 & 0.4815316264 & 0.144918173 \\ 0.9314574825 & -0.0459170889 & 0.6095698017 & 0.3808324602 & 0.8653867983 \\ -0.2372380836 & 0.5190691678 & 0.816552622 & -0.1493625727 & 0.1461689364 \end{bmatrix}$$

resulting in the difference between two matrices equal to

$$|\mathbf{T} - \mathbf{T}_{ori}| = \begin{bmatrix} 0.000000114 & 0.0000005176 & 0.0000001806 & 0.0000003711 & 0.0000003973 \\ 0.0000001809 & 0.0000003766 & 0.000000029 & 0.0000002049 & 0.0000002191 \\ 0.0000004731 & 0.0000000408 & 0.0000003384 & 0.0000001182 & 0.0000004433 \\ 0.0000001914 & 0.0000001749 & 0.0000006844 & 0.0000001807 & 0.0000003591 \\ 0.0000005783 & 0.0000003923 & 0.0000008302 & 0.0000005924 & 0.0000007123 \end{bmatrix}.$$

6 Evaluation

To evaluate our verification approach described in Section 4 and Section 5, we compared it to our earlier work based on invariant checking [23].

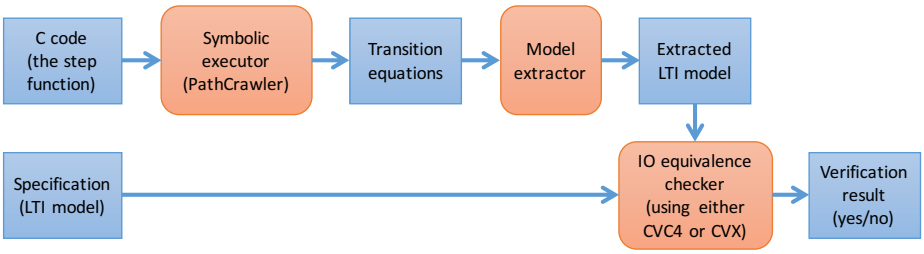

Fig. 1. The verification toolchain for the similarity checking-based approach.

6.1 Verification Toolchain

We implemented an automatic verification framework (presented in Fig. 1) based on the proposed approach described in Section 4 and Section 5. We refer to this approach as similarity checking (SC)-based approach. Given a step function (i.e., C code), we employ the off-the-shelf symbolic execution tool PathCrawler [32] to symbolically execute the step function and generate a set of transition equations. The model extractor which implements the method in Section 4.2 extracts an LTI model from the transition equations. Finally, the equivalence checker based on the method in Section 5 decides the similarity between the extracted LTI model and the given specification (i.e., LTI model), and produces the verification result. The equivalence checker uses either the SMT solver CVC4 [4][6] or the convex optimization solver CVX [14] depending on the formulation employed, which is described in Section 5.

For the invariant checking (IC)-based approach described in Section 3, we use the toolchain Frama-C/Why3/Z3 to verify C code with annotated controller invariants [23]. The step function is annotated with the invariants as described in Section 3. Given annotated C code, Frama-C/Why3 [9, 5] generates proof obligations as SMT instances. The SMT solver Z3 [11][7] solves the proof obligations and produces the verification result (see [23] for more details).

6.2 Scalability Evaluation

To evaluate the SC-based approach compared to the IC-based approach, we randomly generate stable linear controller specifications (i.e., the elements of $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$). Since we observed that the controller dimension n dominates the performance (i.e., running time) of both approaches, we vary n from 2 to 14, and generate three controller specifications for each n. For each controller specification, we employ the code generator Embedded Coder to generate the step function in C. Since we use the LTI system block of Simulink for code generation, the structure of generated C code is not straightforward, having multiple

[6] CVC4 was chosen among other SMT solvers because it showed the best performance for our QF_LRA SMT instances.

[7] Z3 was chosen among other SMT solvers because it showed the best performance for the generated proof obligations in our experiment.

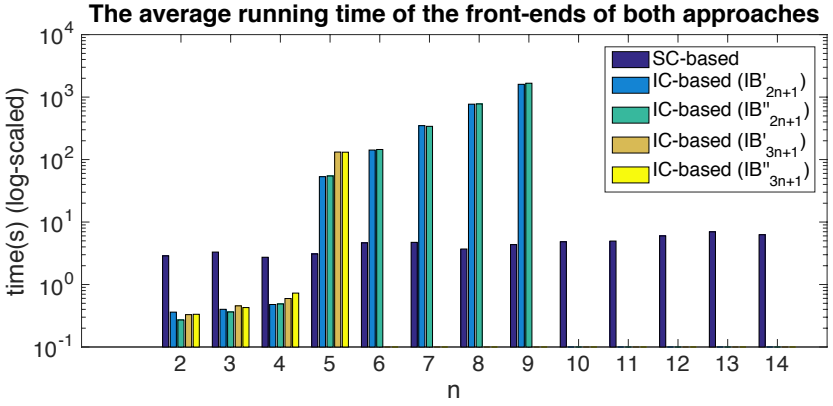

Fig. 2. The average running time of the front-ends of both SC-based and IC-based approaches (with the log-scaled y-axis)

loops and pointer arithmetic operations as illustrated in the step function [24]. This negatively affects the performance of the IC-based approach for reasons to be described later in this subsection. For a comparative evaluation, we use both SC-based and IC-based approaches to verify the generated step function C code against its specification. For each generated controller, we checked that IC-based and SC-based approaches give the same verification result, as long as both complete normally.

To thoroughly compare both approaches, we measure the running time of the front-end and the back-end of each approach separately. By the front-end, we refer to the process from parsing C code to generating proof obligations to be input for constraint solvers. The front-end of the SC-based approach includes the symbolic execution by PathCrawler and the model extraction, while the front-end of the IC-based approach is processing annotated code and generating proof obligations by Frama-C/Why3. On the other hand, by the back-end, we refer to the process of constraint solving. While the back-end of the SC-based approach is the IO equivalence checking based on either SMT solving using CVC4 or convex optimization solving using CVX, the back-end of the IC-based approach is proving the generated proof obligations using Z3.

We first evaluate the frond-end of both approaches (i.e., the whole verification process until constraint solving). Fig. 2 shows that the average running time of the front-ends of both approaches, where missing bars indicate no data due to the lack of scalability of the utilized verification approach (e.g., the tool's abnormal termination or no termination for a prolonged time). Here, IB'_{2n+1}, IB''_{3n+1}, IB''_{3n+1} and IB'_{2n+1} denote the variations of annotating methods as described in [23]. We observe that the running time of the IC-based approaches exponentially increase as the controller dimension n increases, while the SC-based approach remains scalable. The main reason for this is that the IC-based approach requires the preprocessing of code [23], which is unrolling the execution of the step function multiple times (e.g., $2n + 1$ or $3n + 1$ times) as well as

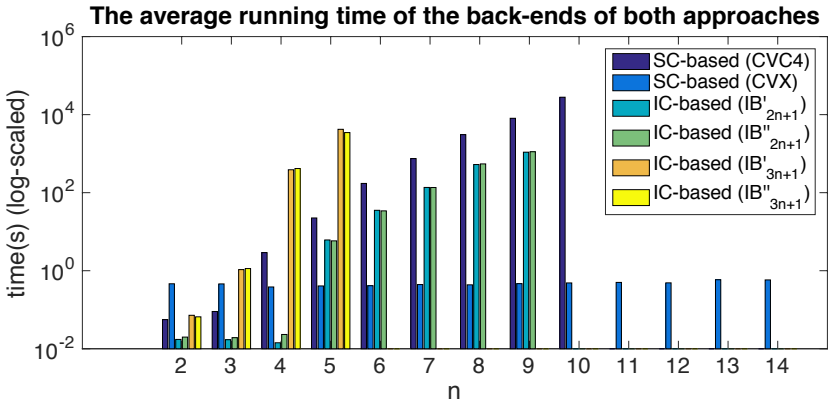

Fig. 3. The average running time of the back-ends of both SC-based and IC-based approaches (with the log-scaled y-axis)

unrolling each loop in the step function $(n+1)$ times. Therefore, in contrast with the SC-based approach, the IC-based approach needs to handle the significantly increased lines of code due to unrolling, so it does not scale up.

Next, we evaluate the back-end of both approaches (i.e., constraint solving). Fig. 3 shows the average running time of the back-ends of both approaches, where missing bars result from the lack of scalability of either the constraint solver used at this stage or the front-end tools. "SC-based (CVC4)" denotes the SMT-based formulation while "SC-based (CVX)" denotes the convex optimization-based formulation. Recall that the SC-based approach using CVC4 and the IC-based approaches employ the SMT solvers for constraint solving, which uses the arbitrary-precision arithmetic. We observe that the running time of the back-ends of those approaches exponentially increase as the controller dimension n increases because of the cost of the bignum arithmetic, while the SC-based approach using CVX remains scalable.

7 Related Work

Recently, there has been much attention to research on high-assurance control software for cyber physical systems (e.g., [28, 1, 21, 20, 19, 10, 12]). First of all, there has been a line of work focused on robust controller software implementations. For example, in [28], a model-based simulation platform is presented to analyze controllers' robustness. In [1, 21], the authors present a fixed-point design method for robust, stable, error-minimized controller implementations. [19] presents a robustness analysis tool to analyze the uncertainties of measurements and plant states. In [10, 12], the authors address the synthesis of fixed-point controller software using SMT solvers. Moreover, there exists work on verifying the control-related properties of Simulink models using theorem proving [2]. Yet, the verification is done at the model level, not at the code level.

However, there has been less attention given to the code-level verification of controller software. In [27, 20], the authors present equivalence checking between

Simulink diagrams and generated code. Yet, they are based on the compliance of the structures between Simulink models and code, instead of observational equivalence checking. In addition, there is a closely related work based on the concept of proof-carrying code for control software [13, 15, 31, 30]. The authors propose the code annotations for control-related properties based on Lyapunov functions, and introduce the PVS linear algebra libraries [15] to verify the properties. However, their focus is limited to only stability and convergence properties rather than the correctness of controller implementation against its model. Moreover, their approaches require the control of code generators, which may introduce intellectual property concerns. Our own earlier work [23] presents a method to verify the correctness of controller implementations by annotating the controllers' invariants. However, the scalability of this method is challenged for real controller implementations with large state dimensions.

Finally, the model extraction technique has been used in software verification [8, 16, 17, 29, 25]. The authors in [8, 16, 17] extract finite state models from implementations to facilitate software model checking. [29] and [25] apply the symbolic execution technique to implemented source code to extract mathematical functional models and high-level state machine models, respectively.

8 Conclusion

We have presented an approach for the verification of linear controller implementations against mathematical specifications. By this, a higher degree of assurance for generated control code can be provided without trusting a code generator. We have proposed to use the symbolic execution technique to reconstruct mathematical models from linear time-invariant controller implementations. We have presented a method to check input-output equivalence between the specification model and the extracted model using the SMT formulation and the convex optimization formulation. Through the evaluation using randomly generated specification and code by Matlab, we showed that the scalability of our new approach has significantly improved compared to our own eariler work. Future work includes the analysis of the effect of floating-point calculations in control code.

Acknowledgments. This work was supported in part by NSF CNS-1505799, NSF CNS-1505701, and the Intel-NSF Partnership for Cyber-Physical Systems Security and Privacy. This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0247. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This research was supported in part by Global Research Laboratory Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2013K1A1A2A02078326) with DGIST.

References

1. Anta, A., Majumdar, R., Saha, I., Tabuada, P.: Automatic verification of control system implementations. In: Proc. 10th ACM International Conference on Embedded Software. pp. 9–18. EMSOFT'10 (2010)
2. Araiza-Illan, D., Eder, K., Richards, A.: Formal verification of control systems' properties with theorem proving. In: UKACC International Conference on Control (CONTROL). pp. 244–249 (2014)
3. Aström, K.J., Murray, R.M.: Feedback systems: an introduction for scientists and engineers. Princeton university press (2010)
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: Computer aided verification. pp. 171–177. Springer (2011)
5. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64 (2011)
6. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Software Testing, Verification and Reliability 16(2), 97–121 (2006)
7. Clarke, L.: A system to generate test data and symbolically execute programs. Software Engineering, IEEE Transactions on (3), 215–222 (1976)
8. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Bby, R., Zheng, H.: Bandera: Extracting finite-state models from java source code. In: Software Engineering, 2000. Proceedings of the 2000 International Conference on. pp. 439–448. IEEE (2000)
9. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c. In: Software Engineering and Formal Methods, pp. 233–247 (2012)
10. Darulova, E., Kuncak, V., Majumdar, R., Saha, I.: Synthesis of fixed-point programs. In: Proc. 11th ACM International Conference on Embedded Software. pp. 22:1–22:10. EMSOFT'13 (2013)
11. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340 (2008)
12. Eldib, H., Wang, C.: An SMT based method for optimizing arithmetic computations in embedded software code. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 33(11), 1611–1622 (2014)
13. Feron, E.: From control systems to control software. Control Systems, IEEE 30(6), 50–71 (2010)
14. Grant, M., Boyd, S.: CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx (Mar 2014)
15. Herencia-Zapana, H., Jobredeaux, R., Owre, S., Garoche, P.L., Feron, E., Perez, G., Ascariz, P.: PVS linear algebra libraries for verification of control software algorithms in C/ACSL. In: NASA Formal Methods, pp. 147–161 (2012)
16. Holzmann, G.J., H Smith, M.: Software model checking: extracting verification models from source code. Software Testing, Verification and Reliability 11(2), 65–79 (2001)
17. Holzmann, G.J., Smith, M.H.: An automated verification method for distributed systems software based on model extraction. Software Engineering, IEEE Transactions on 28(4), 364–377 (2002)
18. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)

19. Majumdar, R., Saha, I., Shashidhar, K., Wang, Z.: CLSE: Closed-loop symbolic execution. In: NASA Formal Methods, pp. 356–370 (2012)
20. Majumdar, R., Saha, I., Ueda, K., Yazarel, H.: Compositional equivalence checking for models and code of control systems. In: 52nd Annual IEEE Conference on Decision and Control (CDC). pp. 1564–1571 (2013)
21. Majumdar, R., Saha, I., Zamani, M.: Synthesis of minimal-error control software. In: Proc. 10th ACM International Conference on Embedded Software. pp. 123–132. EMSOFT'12 (2012)
22. Mathworks: Bug Reports for Incorrect Code Generation, http://www.mathworks.com/support/bugreports/?product=ALL&release=R2015b&keyword=Incorrect+Code+Generation
23. Pajic, M., Park, J., Lee, I., Pappas, G.J., Sokolsky, O.: Automatic verification of linear controller software. In: 12th International Conference on Embedded Software (EMSOFT). pp. 217–226. IEEE Press (2015)
24. Park, J.: Step function example, http://dx.doi.org/10.5281/zenodo.44338
25. Pichler, J.: Specification extraction by symbolic execution. In: Reverse Engineering (WCRE), 2013 20th Working Conference on. pp. 462–466. IEEE (2013)
26. Rugh, W.J.: Linear system theory. Prentice Hall (1996)
27. Ryabtsev, M., Strichman, O.: Translation validation: From simulink to c. In: Computer Aided Verification. pp. 696–701. Springer (2009)
28. Sangiovanni-Vincentelli, A., Di Natale, M.: Embedded system design for automotive applications. IEEE Computer (10), 42–51 (2007)
29. Wang, S., Dwarakanathan, S., Sokolsky, O., Lee, I.: High-level model extraction via symbolic execution. Technical Reports (CIS) Paper 967, University of Pennsylvania, http://repository.upenn.edu/cis_reports/967 (2012)
30. Wang, T., Jobredeaux, R., Herencia, H., Garoche, P.L., Dieumegard, A., Feron, E., Pantel, M.: From design to implementation: an automated, credible autocoding chain for control systems. arXiv preprint arXiv:1307.2641 (2013)
31. Wang, T.E., Ashari, A.E., Jobredeaux, R.J., Feron, E.M.: Credible autocoding of fault detection observers. In: American Control Conference (ACC). pp. 672–677 (2014)
32. Williams, N., Marre, B., Mouy, P., Roger, M.: Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In: Dependable Computing-EDCC 5, pp. 281–292. Springer (2005)