Automatic Verification of Finite Precision Implementations of Linear Controllers

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

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

Abstract. We consider the problem of verifying finite precision implementation of linear time-invariant controllers against mathematical specifications. A specification may have multiple correct implementations which are different from each other in controller state representation, but equivalent from a perspective of input-output behavior (e.g., due to optimization in a code generator). The implementations may use finite precision computations (e.g. floating-point arithmetic) which cause quantization (i.e., roundoff) errors. To address these challenges, we first extract a controller's mathematical model from the implementation via symbolic execution and floating-point error analysis, and then check approximate input-output equivalence between the extracted model and the specification by similarity checking. We show how to automatically verify the correctness of floating-point controller implementation in C language using the combination of techniques such as symbolic execution and convex optimization problem solving. We demonstrate the scalability of our approach through evaluation with randomly generated controller specifications of realistic size.

1 Introduction

Most modern safety- and life-critical embedded applications rely on software-based control for their operation. When reasoning about safety of these systems, it is extremely important to ensure that control software is correctly implemented. In this paper, we study the problem of whether a given piece of software is a faithful representation of an abstract specification of the control function.

We assume a commonly used development approach, where control systems are developed in a model-driven fashion. The model captures both the dynamics of the "plant", the entity to be controlled, and the controller itself, as mathematical expressions using well established tools, such as Simulink and Stateflow. Control theory offers a rich set of techniques to determine these expressions, determine their parameters, and perform analysis of the model to conclude whether the plant model adequately describes the system to be controlled and

A. Legay and T. Margaria (Eds.): TACAS 2017, Part I, LNCS 10205, pp. 153–169, 2017.
DOI: 10.1007/978-3-662-54577-5_9

whether the controller achieves the desired goals of the control system. In this work, we assume that such control design activities have been performed, achieving the acceptable degree of assurance for the control design. In other words, we assume that the mathematical model of the controller is correct with respect to any higher-level requirements and can be used as the specification for a software implementation of the controller.

Typically, control software is obtained by code generation from the mathematical model. Code generation tools such as Embedded Coder are widely used. Ideally, we would like to have these code generation tools to be verified, that is, to offer guarantees that generated code correctly computes the control function. In this case, no verification of the control code would be needed. However, commercially available code generators are complex black-box tools and are generally not amenable to formal verification. Subtle bugs have been found in commercially available code generators in the past [25]. In the absence of verified code generators, we would like to be able to verify instances of generated code with respect to their mathematical specification.

In our past work [26,28], we explored several approaches to the verification of implementations of linear time invariant (LTI) controllers. In LTI controllers, the relationships between the values of inputs and state variables, and between state variables and outputs, are captured as linear functions, and coefficients of these functions are constant (i.e., time-invariant). The main limitation in all of these approaches is the assumption that the calculations are performed using real numbers. Of course, real numbers are a mathematical abstraction. In practice, software performs calculations using a limited-precision representation of numbers, such as the floating-point representation. The use of floating-point numbers introduces errors into the computation, which have to be accounted for in the verification process.

In this paper, we build on the work of [28], which follows an *equivalence checking* approach. We apply symbolic execution to the generated code, which calculates symbolic expressions for the values of state and output variables in the code at the completion of the invocation of the controller. We use these symbolic values to reconstruct a mathematical representation of the control function. We introduce error terms into this representation that characterize the effects of numerical errors. The verification step then tries to establish the approximate equivalence between the specification of the control function and the reconstructed representation. In [28], we considered two promising alternatives for assessing the equivalence: one based on SMT solving and the other one based on convex optimization. Somewhat surprisingly, when the error terms that account for floating-point calculations are added, the SMT-solving approach becomes impractical, while the optimization-based approach suffers minimal degradation in performance.

The paper is organized as follows: Sect. 2 provides background of LTI systems. Section 3 describes how to extract a model from the controller code. Section 4 presents the approximate equivalence checking. Section 5 evaluates the scalability of our approach. Sections 6 and 7 provide an overview of related work and conclude the paper.

2 Preliminaries

This section presents preliminaries on LTI controllers and their software implementations. We also introduce two real-world examples that motivate the problem considered in this paper, as well as the problem statement.

2.1 Linear Feedback Controller

The goal of feedback controllers is to ensure that the closed-loop systems have certain desired behaviors. In general, these controllers derive suitable control inputs to the plants (i.e., systems to control) based on previously obtained measurements of the plant outputs. In this paper, we consider a general class of feedback controllers that can be specified as linear time-invariant (LTI) controllers 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 $\mathbf{u}_k \in \mathbb{R}^p$, $\mathbf{y}_k \in \mathbb{R}^m$ and $\mathbf{z}_k \in \mathbb{R}^n$ are the input vector, the output vector and the state vector at time k respectively. 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 use $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{z}_0)$ to denote an LTI controller, or just use $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ when the initial controller state $\mathbf{z}_0$ is zero.

During the control-design phase, controller $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{z}_0)$ is derived to guarantee the desired closed-loop performance, while taking into account available computation and communication resources (e.g., finite-precision arithmetic logic units). This model (i.e., controller specification) is then usually 'mapped' into a software implementation of a *step function* that: (1) maintains the state of the controller, and updates it every time new sensor measurements are available (i.e., when it's invoked); and (2) computes control outputs (i.e., inputs applied to the plant) from the current controller's state and incoming sensor measurements. In most embedded control systems, the step function is periodically invoked, or whenever new sensor measurements arrive. In this work, as in our previous work [28], we assume that data is exchanged with the step function through global variables.[1] 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. It is worth noting however 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.

2.2 Motivating Examples

To motivate our work, we introduce two examples from [26,28]. These examples illustrate limitations of the standard verification techniques that directly utilize

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

the mathematical model from (1), in cases when controller software is generated by a code generator whose optimizations potentially violate the model while still ensuring the desired control functionality.

A Scalar Linear Integrator. Consider a simple controller that computes control input u_k as a scaled sum of all previous sensor data $y_i \in \mathbb{R}, i = 0, ..., k-1$ – i.e.,

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

Now, if we use the Simulink Integrator block with Forward Euler integration to implement this controller, the resulting controller model will be $\Sigma(1, \alpha, 1, 0)$, – i.e., $z_{k+1} = z_k + \alpha y_k, u_k = z_k$. On the other hand, a realization $\hat{\Sigma}(1, 1, \alpha, 0)$ – i.e., $z_{k+1} = z_k + y_k, u_k = \alpha z_k$, of the controller would introduce a reduced computational error when finite precision arithmetics is used [10]. Thus, controller specification (2) may result in two different software implementations due to the use of different code generation tools. Still, it is important to highlight that these two implementations would have identical input-output behavior – the only difference is whether they maintain a scaled or unscaled sum of the previous sensor measurements.

Multiple-Input-Multiple-Output Controllers. Now, consider a more general Multiple-Input-Multiple-Output (MIMO) controller with two inputs and two outputs which maintains five states

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

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

The controller has to perform $25 + 10 = 35$ multiplications as part of the state $\mathbf{z}$ update in every invocation of the step function. On the other hand, the following controller requires only $5 + 10 = 15$ multiplications for state update.

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

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

The above controllers Σ and $\hat{\Sigma}$ are *similar*,[2] which means that the same input sequences $\mathbf{y}_k$ delivered to both controllers, would result in identical outputs of the controllers. Note that the controller's states will likely differ. Consequently, the 'diagonalized' controller $\hat{\Sigma}$ results in the same control performance and thus provides the same control functionality as Σ, while violating the state evolution model of the initial controller Σ. The motivation for the use of the diagonalized controller comes from a significantly reduced computational cost that allow for the utilization of resource constrained embedded platforms. In general, any controller (1), would require $n^2 + np = n(n+p)$ multiplications to update its state. This can be significantly reduced when matrix $\mathbf{A}$ in (1) is diagonal – in this case only $n + np = n(p+1)$ multiplications are needed.

2.3 Problem Statements

As illustrated with the motivating examples, the initial controller model and its implementation (i.e., step function) may be different from each other in representation (i.e., controller parameters, state representation) due to the optimization of code generators, while being functionally equivalent from the input-output perspective. Thus, we would like to develop the verification technique that is not sensitive to the state representation of the controller. Moreover, the controller software for embedded systems uses a finite precision arithmetic which introduces rounding errors in the computation. Thus, it is necessary to analyze the effect of rounding errors in the verification process. This work focuses on the controller implementations using floating-point arithmetic, but can be easily extended to the setup for fixed-point arithmetic. Consequently, our problem statements are as follows: given an LTI model, a step function using floating-point arithmetic and an approximate equivalence precision, verify if the step

[2] Similarity transform is formally defined in Sect. 4.

function is approximately equivalent to the initial LTI model from the input-output perspective.

3 Extracting Model from Floating-Point Controller Implementation

Our approach to the verification of a controller implementation against its mathematical model takes two steps: we first extract a model from the finite precision implementation (i.e., step function using floating-point arithmetic), and then compare it with the original model. This approach is an extension of [28] to consider the quantization error in the finite-precision implementation. To obtain a model from the step function, we employ the symbolic execution technique [7,21], which allows us to identify the computation of the step function (i.e., the big-step transition relation on global states between before and after the execution of the step function). From the transition relation, we extract a mathematical model for the controller implementation. Since the implementation has floating-point quantization (i.e., roundoff) errors, the representation of the extracted model includes roundoff error terms, thus being different from the representation of the initial LTI model (1). We will describe the representation of extracted models in the next subsection.

3.1 Quantized Controller Model

A finite precision computation (e.g., floating-point arithmetic) involves rounding errors, which makes the computation result slightly deviated from the exact value that might be computed with the infinite precision computation. The floating-point rounding error can be modeled with the notions of both *absolute error* and *relative error*. The absolute error is defined as the difference between an exact number and its rounded number. The relative error defines such difference relative to the exact number. To model quantized controller implementations, we extend the representation of LTI model (1) with the new terms of absolute errors and relative errors, and obtain the following representation of quantized controller model:

$$\begin{aligned} \hat{\mathbf{z}}_{k+1} &= (\hat{\mathbf{A}} + \mathbf{E_A})\hat{\mathbf{z}}_k + (\hat{\mathbf{B}} + \mathbf{E_B})\mathbf{u}_k + \mathbf{e_z} \\ \mathbf{y}_k &= (\hat{\mathbf{C}} + \mathbf{E_C})\hat{\mathbf{z}}_k + (\hat{\mathbf{D}} + \mathbf{E_D})\mathbf{u}_k + \mathbf{e_y}. \end{aligned} \tag{7}$$

where $\hat{\mathbf{A}}$, $\hat{\mathbf{B}}$, $\hat{\mathbf{C}}$ and $\hat{\mathbf{D}}$ are controller parameters. $\mathbf{E_A}$, $\mathbf{E_B}$, $\mathbf{E_C}$ and $\mathbf{E_D}$ are the relative errors regarding the state and input variables which are bounded by the relative error bound b_{rel} such that $\|\mathbf{E_A}\|, \|\mathbf{E_B}\|, \|\mathbf{E_C}\|, \|\mathbf{E_D}\| \leq b_{rel}$ where $\|\cdot\|$ is the L_∞ norm operator. In addition, $\mathbf{e_z}$ and $\mathbf{e_y}$ are the absolute errors which are bounded by the absolute error bound b_{abs} such that $\|\mathbf{e_x}\|, \|\mathbf{e_y}\| \leq b_{abs}$. In the rest of this section, we explain how to extract a quantized controller model $(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \hat{\mathbf{D}}, b_{rel}, b_{abs})$ from the floating-point controller implementation via symbolic execution and floating-point error analysis techniques.

3.2 Symbolic Execution of Floating-Point Controller Implementation

In our approach, the symbolic execution technique [7,21] is employed to analyze the step function C code. We symbolically execute the step function with symbolic values such as symbolic inputs and symbolic controller states, and examine the change of the program's global state where the output and new controller state are updated with symbolic expressions in terms of the symbolic values. The goal of the symbolic execution in our approach is to find symbolic formulas that concisely represent the computation of the step function C code that originally has loops and pointer arithmetic operations. The idea behind this symbolic execution process is that the linear controller implementations that we consider in this work have simple control flows for the sake of deterministic real-time behaviors (e.g., fixed upper bound of loops), thus being amenable to our symbolic execution process. Consequently, the symbolic execution of linear controller implementations yield finite and deterministic symbolic execution paths [28].

However, unlike [28], our work herein newly considers the effect of floating-point rounding errors in the step function. Thus it is necessary to pay special attention (e.g., normalization [6]) to the floating-point computation in symbolic execution. When symbolic expressions are constructed with floating-point operators in the course of symbolic execution, the evaluation order of floating-point operations should be preserved according to the floating-point program semantics, because floating-point arithmetic does not hold basic algebraic properties such as associativity and distributivity in general.

Once the symbolic execution is completed, symbolic formulas are produced. The symbolic formulas represent the computation of the step function in a concise way (i.e., in the arithmetic expression form without loops, function calls and side effects). The produced symbolic formula has the following form, which we call *transition equation*:

$$v^{(new)} = f(v_1, v_2, \ldots, v_t) \tag{8}$$

where $v^{(new)}$ is a global variable which is updated with the symbolic expression, v_i are the initial symbolic values of the corresponding variables before the symbolic execution of the step function. $f(v_1, v_2, \ldots, v_t)$ is the symbolic expression that consists of floating-point operations where t is the number of variables used in f. This expression should preserve the correct order of evaluation according to the floating-point semantics of the step function C code.

For example, consider the step function in [27], which is generated by Embedded Coder (the code generator of MATLAB/Simulink) for the LTI controller models (5) and (6). We illustrate one of the transition equations obtained from the symbolic execution of the step function as follows:

$$\begin{aligned} \mathtt{y}[1]^{(new)} = (((((&0.503767 \otimes \mathtt{x}[0]) \oplus (-0.573538 \otimes \mathtt{x}[1])) \oplus (0.170245 \otimes \mathtt{x}[2])) \\ &\oplus(-0.583312 \otimes \mathtt{x}[3])) \oplus (-0.56603 \otimes \mathtt{x}[4])). \end{aligned} \tag{9}$$

where `x` is the shortened name for `LTIS_DW.Internal_DSTATE`, and `y` is the shortened name for `LTIS_Y.y` for presentation purposes only, and $\oplus$, $\ominus$ and $\otimes$ are floating-point operators corresponding to $+$, $-$ and $\times$ respectively. In the next

subsection, we explain how to extract the quantized model (17) from the symbolic expressions.

3.3 Quantization Error Analysis and Model Extraction

This subsection explains how to extract the quantized controller model (17) from a set of symbolic expressions (8) obtained from the step function. The symbolic expression consists of floating-point operations of symbolic values and numeric constants. We first describe how to analyze the floating-point quantization (i.e., roundoff) error in the symbolic expression evaluation. Since we only consider linear controller implementations rejecting nonlinear cases in the symbolic execution phase, the symbolic expression f obtained from the step function has the following syntax, thus guaranteeing the linearity:

$$f := v \mid f \oplus f \mid f \ominus f \mid f \circledast f_c \mid f_c \circledast f$$
$$f_c := c \mid f_c \circledast f_c$$

where v is a variable (i.e., the initial symbolic value of the variable), c is a constant, and $\circledast \in \{\oplus, \ominus, \otimes\}$. f_c is a sub-expression which contains no variable, thus being evaluated to a constant, while f contains at least one variable. The multiplication operation $\otimes$ appears only when at least one operand is a constant-expression f_c, thus preventing the expression from being nonlinear (i.e., the product of two symbolic values).

In order to simplify a certain program analysis problem, a common assumption is often made in the literature [14,28] that the floating-point operations (e.g., $\oplus$, $\ominus$ and $\otimes$) behave the same way as the real operations (e.g., $+$, $-$ and $\times$) with no rounding. Under this assumption, the Eq. (8) can be represented in the following canonical form [28]:

$$v^{(new)} = \sum_{i=1}^{t} c_i v_i \tag{10}$$

where t is the number of product terms, v, v_i are variables, and c_i is the coefficient. In reality, however, floating-point numbers have limited precision, and the floating-point operations involve rounding errors. In this work, we consider the effect of such floating-point rounding errors in the verification.

The IEEE 754 standard [1] views a finite precision floating-point operation as the corresponding real operation followed by a rounding operation:

$$x_1 \circledast x_2 = rnd(x_1 * x_2) \tag{11}$$

where $\circledast \in \{\oplus, \ominus, \otimes\}$ and $*$ is the corresponding real arithmetic operation to $\circledast$. A rounding operator rnd is a function that takes a real number as input and returns as output a floating-point number that is closest to the input real number, thus causes a quantization error (i.e., rounding error) in the floating-point operation. There are multiple common rounding operators (e.g. round to the nearest, ties to even) defined in the IEEE 754 standard [1]. A rounding operator can be modeled as follows [15]:

$$rnd(x) = x(1+e) + d \tag{12}$$

for some e and d where e is a relative error, d is an absolute error, and $|e| \leq \epsilon$ and $|d| \leq \delta$. ϵ and δ can be determined according to the rounding mode and the precision (i.e., the number of bits) of the system. For example, $\epsilon = 2^{-53}$ and $\delta = 2^{-1075}$ for the double precision (i.e., 64 bits) rounding to the nearest [33]. Combining the two Eqs. (11) and (12), we have the following model for the floating-point operations:

$$x_1 \circledast x_2 = (x_1 * x_2)(1+e) + d \tag{13}$$

After rewriting the symbolic expression of the transition equation (8) applying the Eq. (13), suppose that we have the following equation form:

$$v^{(new)} = \sum c_i v_i + err_{rel} + err_{abs} \tag{14}$$

where $\sum c_i v_i$ is the exact expression as (10), and err_{abs} is the absolute error term bounded by b_{abs} such that $|err_{abs}| \leq b_{abs}$. err_{rel} is the relative error term which is related to the variables $\{v_i\}$ (i.e., symbolic values). We rewrite err_{rel} as $\sum err_i v_i$ where err_i is the relative error term specific to the variable v_i, and b_i is the upper bound for err_i such that $|err_i| \leq b_i$. We relax the equation by over-approximating each err_i as follows:

$$\begin{aligned} v^{(new)} &= \sum c_i v_i + \sum err_i v_i + err_{abs} \\ &= \sum c_i v_i + err \sum v_i + err_{abs} \end{aligned} \tag{15}$$

where err is bounded by b_{rel} such that $|err| \leq b_{rel}$ where b_{rel} is defined as $b_{rel} = \max\{b_i\}$.

We now rearrange and group the product terms by variable names such as the state variables and the input variables. We assume that the names of input and output variables are given as the interface of the step function. The state variables can be identified as the variables appearing in the transition equations which are not input variables nor output variables. In addition to the rearrangement, by transforming the sum of products into a form of scalar product of vectors, we have:

$$\begin{aligned} v^{(new)} =& [c_1, c_2, ..., c_n]\mathbf{x} + [err, err, ..., err]\mathbf{x} \\ &+ [c'_1, c'_2, ..., c'_p]\mathbf{u} + [err, err, ..., err]\mathbf{u} + err_{abs} \end{aligned} \tag{16}$$

where $\mathbf{x}$ is the vector of state variables, and $\mathbf{u}$ is the vector of input variables.

Finally, we rewrite the transition equations as two matrix equations as follows:

$$\begin{aligned} \mathbf{x}^{(new)} &= (\hat{\mathbf{A}} + \mathbf{E_A})\mathbf{x} + (\hat{\mathbf{B}} + \mathbf{E_B})\mathbf{u} + \mathbf{e_x} \\ \mathbf{y}^{(new)} &= (\hat{\mathbf{C}} + \mathbf{E_C})\mathbf{x} + (\hat{\mathbf{D}} + \mathbf{E_D})\mathbf{u} + \mathbf{e_y}. \end{aligned} \tag{17}$$

where $\hat{\mathbf{A}} \in \mathbb{R}^{n\times n}$, $\hat{\mathbf{B}} \in \mathbb{R}^{n\times p}$, $\hat{\mathbf{C}} \in \mathbb{R}^{m\times n}$ and $\hat{\mathbf{D}} \in \mathbb{R}^{m\times p}$. The matrices for the relative errors are bounded by b^*_{rel} such that $\|\mathbf{E_A}\|, \|\mathbf{E_B}\|, \|\mathbf{E_C}\|, \|\mathbf{E_D}\| \leq b^*_{rel}$. The absolute error vectors $\mathbf{e_x}$ and $\mathbf{e_y}$ are bounded by b^*_{abs} such that $\|\mathbf{e_x}\|, \|\mathbf{e_y}\| \leq b^*_{abs}$. Note that b^*_{rel} and b^*_{abs} can be easily determined using b_{rel} and b_{abs} obtained from the floating-point error analysis for each transition equation.

For example, consider the transition equation (9), from which via the floating-point error analysis, we have:

$$\begin{aligned}
\mathtt{y}[1]^{(new)} &= (((((0.503767 \otimes \mathtt{x}[0]) \oplus (-0.573538 \otimes \mathtt{x}[1])) \oplus (0.170245 \otimes \mathtt{x}[2])) \\
&\quad \oplus (-0.583312 \otimes \mathtt{x}[3])) \oplus (-0.56603 \otimes \mathtt{x}[4])) \\
&= 0.503767 \cdot \mathtt{x}[0] + -0.573538 \cdot \mathtt{x}[1] + 0.170245 \cdot \mathtt{x}[2] \\
&\quad + -0.583312 \cdot \mathtt{x}[3] + -0.56603 \cdot \mathtt{x}[4] + err_{rel} + err_{abs} \qquad (18) \\
&= 0.503767 \cdot \mathtt{x}[0] + -0.573538 \cdot \mathtt{x}[1] + 0.170245 \cdot \mathtt{x}[2] \\
&\quad + -0.583312 \cdot \mathtt{x}[3] + -0.56603 \cdot \mathtt{x}[4] \\
&\quad + err(\mathtt{x}[0] + \mathtt{x}[1] + \mathtt{x}[2] + \mathtt{x}[3] + \mathtt{x}[4]) + err_{abs}
\end{aligned}$$

where $|err| \leq \frac{988331}{250000}\epsilon \div (1-4\epsilon) = b_{rel}$, and $|err_{abs}| \leq 4 \cdot (1+\epsilon)^4 \cdot \delta = b_{abs}$. For the double precision (i.e., 64 bits) rounding to nearest (i.e., $\epsilon = 2^{-53}$ and $\delta = 2^{-1075}$), $b_{rel} \approx 4.389071 \times 10^{-16}$ and $b_{abs} \approx 1.235164 \times 10^{-323}$.

4 Approximate Input-Output Equivalence Checking

In order to verify a finite precision implementation of the linear controller, the previous section described how to extract the quantized controller model from the implementation. In this section, we introduce how to compare the extracted model (17) and the initial model (1) with a notion of approximate input-output (IO) equivalence.

4.1 Approximate Input-Output Equivalence

This subsection defines an approximate IO equivalence relation, inspired by the similarity transformation of LTI systems [30]. In order for two LTI systems to be IO equivalent to each other, there must exist an invertible linear mapping $\mathbf{T}$ from one system's state $\mathbf{z}$ to another system's state $\hat{\mathbf{z}}$ such that $\mathbf{z} = \mathbf{T}\hat{\mathbf{z}}$ and $\hat{\mathbf{z}} = \mathbf{T}^{-1}\mathbf{z}$. The matrix $\mathbf{T}$ is referred to as the *similarity transformation matrix* [30]. Assuming that a proper $\mathbf{T}$ is given, we substitute $\mathbf{z}_k$ by $\mathbf{T}\hat{\mathbf{z}}$ in the initial LTI model (1), thus having:

$$\mathbf{T}\hat{\mathbf{z}}_{k+1} = \mathbf{AT}\hat{\mathbf{z}}_k + \mathbf{B}\mathbf{u}_k, \quad \mathbf{y}_k = \mathbf{CT}\hat{\mathbf{z}}_k + \mathbf{D}\mathbf{u}_k.$$

or

$$\hat{\mathbf{z}}_{k+1} = (\mathbf{T}^{-1}\mathbf{AT})\hat{\mathbf{z}}_k + (\mathbf{T}^{-1}\mathbf{B})\mathbf{u}_k, \quad \mathbf{y}_k = (\mathbf{CT})\hat{\mathbf{z}}_k + \mathbf{D}\mathbf{u}_k. \qquad (19)$$

By the similarity transformation, two LTI systems (1) and (19) are *similar*, meaning that they are IO equivalent. We now compare the transformed initial

LTI model (19) and the quantized controller model (17) that is extracted from the step function. Equating the corresponding coefficient matrices of the two models (19) and (17), we have:

$$\mathbf{T}^{-1}\mathbf{AT} = \hat{\mathbf{A}} + \mathbf{E_A}, \quad \mathbf{T}^{-1}\mathbf{B} = \hat{\mathbf{B}} + \mathbf{E_B}, \quad \mathbf{CT} = \hat{\mathbf{C}} + \mathbf{E_C}, \quad \mathbf{D} = \hat{\mathbf{D}} + \mathbf{E_D}$$

or

$$\mathbf{AT} = \mathbf{T}\hat{\mathbf{A}} + \mathbf{TE_A}, \quad \mathbf{B} = \mathbf{T}\hat{\mathbf{B}} + \mathbf{TE_B}, \quad \mathbf{CT} = \hat{\mathbf{C}} + \mathbf{E_C}, \quad \mathbf{D} = \hat{\mathbf{D}} + \mathbf{E_D} \tag{20}$$

However, the equality of the exact equivalence condition (20) will never hold because of the floating-point error terms (e.g., $\mathbf{E_A}$) and the numerical errors in the implementation's controller parameters (e.g., $\hat{\mathbf{A}}$) due to the optimization of the code generator. To overcome this problem, we define and use an approximate equivalence relation $\approx_\rho$ on matrices such that $\mathbf{M} \approx_\rho \hat{\mathbf{M}}$ if and only if $\left\|\mathbf{M} - \hat{\mathbf{M}}\right\| \leq \rho$ where ρ is a given precision (i.e., threshold for approximate equivalence). Note that the approximate equivalence relation $\approx_\rho$ is not transitive, thus not an equivalence relation unless $\rho = 0$. With $\approx_\rho$ for a precision ρ, the Eq. (20) are relaxed as follows:

$$\mathbf{AT} \approx_\rho \mathbf{T}\hat{\mathbf{A}} + \mathbf{TE_A}, \quad \mathbf{B} \approx_\rho \mathbf{T}\hat{\mathbf{B}} + \mathbf{TE_B}, \quad \mathbf{CT} \approx_\rho \hat{\mathbf{C}} + \mathbf{E_C}, \quad \mathbf{D} \approx_\rho \hat{\mathbf{D}} + \mathbf{E_D} \tag{21}$$

Finally, we say that the initial LTI model (1) and the quantized model (17) extracted from the implementation are approximately IO equivalent with precision ρ if there exists a similarity transformation matrix $\mathbf{T}$ which satisfies (21), and the absolute errors of the floating-point computations are negligible (i.e., $\mathbf{e_z} \approx_\rho \mathbf{0}$ and $\mathbf{e_y} \approx_\rho \mathbf{0}$). Note that the problem of checking the approximate IO equivalence is the problem of finding a proper similarity transformation matrix. In the rest of this section, we explain how to find the similarity transformation matrix using a satisfiability problem formulation and a convex optimization problem formulation.

4.2 Satisfiability Problem Formulation

This section discusses the satisfiability problem formulation for the approximate IO equivalence checking. To find the similarity transformation matrix using existing SMT solvers, the problem can be formulated roughly as follows:

$$\exists \mathbf{T} : \forall \mathbf{E_A}, \mathbf{E_B}, \mathbf{E_C}, \mathbf{E_D} : \|\mathbf{E_A}\|, \|\mathbf{E_B}\|, \|\mathbf{E_C}\|, \|\mathbf{E_D}\| \leq b_{rel} \implies (21) \text{ holds}$$

In this formulation, the variable $\mathbf{T}$ and the relative error variables (e.g., $\mathbf{E_A}$) are quantified alternately, thus requiring exists/forall (EF) problem solving. Moreover, the formula involves the non-linear real arithmetic (NRA) due to the terms $\mathbf{TE_A}$ and $\mathbf{TE_B}$ in (21). For these reasons, the scalability of this SMT formulation-based approach is questionable because the current SMT solvers rarely supports EF-NRA problem solving with scalability. In the next subsection, we describe a more efficient approach based on convex optimization as an alternative method.

4.3 Convex Optimization Formulation

This subsection describes the convex optimization-based approach to the approximate IO equivalence checking. Since the relative error variables $\mathbf{E_A}$ make the condition (21) inappropriate to be formulated as a convex optimization problem, our approach is to derive a sufficient condition for (21). By over-approximating the error terms and removing the error variables, we derive such a sufficient condition for (21) which is formulated as a convex optimization problem as follows:

$$\begin{array}{ll} \text{variables} & e \in \mathbb{R}, \mathbf{T} \in \mathbb{R}^{n \times n} \\ \text{minimize} & e \\ \text{subject to} & \left\| \hat{\mathbf{A}}\mathbf{T} - \mathbf{T}\mathbf{A} \right\|_\infty + n^2 \left\| \mathbf{T} \right\|_\infty b_{rel} \leq e \\ & \left\| \hat{\mathbf{B}} - \mathbf{T}\mathbf{B} \right\|_\infty + n^2 \left\| \mathbf{T} \right\|_\infty b_{rel} \leq e \\ & \left\| \hat{\mathbf{C}}\mathbf{T} - \mathbf{C} \right\|_\infty + n \cdot b_{rel} \leq e, \quad \left\| \hat{\mathbf{D}} - \mathbf{D} \right\|_\infty + n \cdot b_{rel} \leq e \end{array} \tag{22}$$

The idea behind this formulation is to use convex optimization to find the minimum precision e and then check whether $e \leq \rho$ where ρ is the given precision.

Remark 1. Our verification method is sound (i.e., no false positive) but not complete. Due to the relaxations both in the floating-point error approximation and the approximate IO equivalence checking, there might be a case with a model and a correct implementation where our method remains indecisive in the equivalence decision. This can be potentially improved by tightening the relaxations in future work. In addition, a larger ρ can make the approximate equivalence decision positive, which is not with a smaller ρ. The IO equivalence with a large ρ may not guarantee the controller's well-behavedness. Relating the approximate equivalence precision ρ and the performance of the controller (e.g., robustness) is an avenue of future work.

5 Evaluation

This section presents our toolchain for the verification of finite precision controller implementations, and evaluates its scalability. We also evaluate computational overhead (i.e., running time) over our own earlier work [28] which assumes that the computations of controller implementations have no rounding errors.

5.1 Toolchain

This subsection presents the verification toolchain (shown in Fig. 1) that we implemented based on our method described in this paper. The toolchain is an extension of [28] to consider the floating-point error of step function in verification. The toolchain takes as input a step function C code and an LTI model specification. We use the off-the-shelf symbolic execution tool PathCrawler [37] to symbolically execute the step function and produce the transition equations

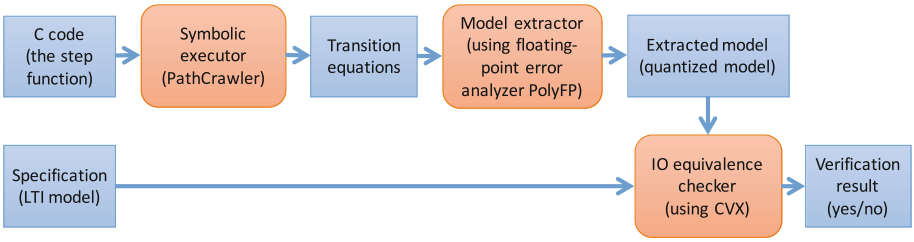

Fig. 1. The verification toolchain

for the step function. From the transition equations, the model extractor based on Sect. 3.3 extracts the quantized controller model using the floating-point error analysis tool PolyFP [2]. Finally, the extracted quantized model is compared with the given specification (i.e., LTI model) based on the approximate IO relation defined in Sect. 4. The approximate IO equivalence checker uses the convex optimization solver CVX [17] to solve the formulas in Sect. 4.3.

5.2 Scalability Analysis

This subsection evaluates the scalability of our approach/toolchain presented in this paper. To evaluate, we use the Matlab function `drss` to randomly generate discrete stable linear controller specifications (i.e., the elements of $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$) varying the controller dimension n from 2 to 14. To obtain an IO equivalent implementation, we perform an arbitrary similarity transformation on $\mathbf{\Sigma}$, and yield the transformed model $\hat{\mathbf{\Sigma}}$. We use an LTI system block of Simulink to allow the Embedded Coder (i.e., code generator of Matlab/Simulink) to generate a floating-point implementation (i.e., step function in C) for $\hat{\mathbf{\Sigma}}$. Note that the generated step function has multiple loops and pointer arithmetic operations as illustrated in the step function in [27]. We employ our toolchain to verify that the generated step function correctly implements the original controller model. We pick the precision ρ to be 10^{-6} to tolerate both numerical errors in the similarity transformation and the floating-point controller implementation.

We now evaluate the scalability of our approach running our toolchain with the random controller specifications and their implementations generated. We measure the running time of the front-end and the back-end of our approach separately. The front-end refers to the process of symbolic execution of the step function (using PathCrawler) and model extraction using the floating-point analysis (using PolyFP). The back-end refers to the approximate IO equivalence checking using convex optimization problem solving (using CVX). The scalability analysis result is shown in Fig. 2, which demonstrates that our approach is scalable for the realistic size of controller dimension.

We now evaluate the overhead of our approach compared to the previous work [28] where the verification problem is simpler than our verification problem herein because the previous work [28] assumes that the computation of step function C code is exact without having any roundoff error. Our approach herein provides a higher assurance for the finite precision controller implementations considering the rounding errors in computation. Figure 3 shows the computational

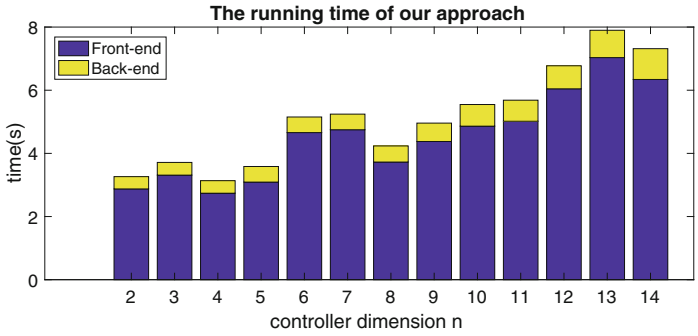

Fig. 2. The running time of both the front-end and the back-end of our approach

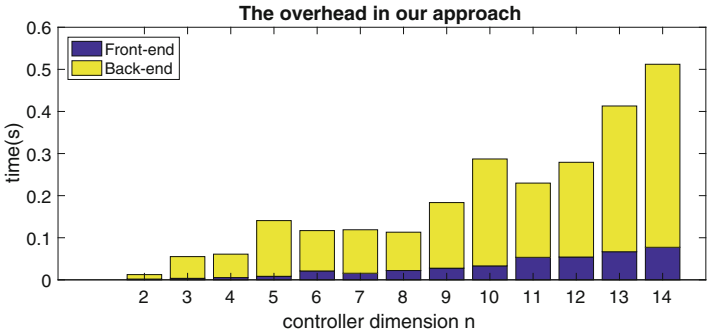

Fig. 3. The overhead in both the front-end and the back-end of our approach

overhead (i.e., the increase of running time) in our approach as a result of considering the floating-point roundoff error in controller implementation verification. We observe that the overhead of the floating-point error analysis in the front-end is marginal. The running time of the back-end increases because the convex optimization problem formulation for approximate IO equivalence requires more computations to solve. Finally, the total running time only increases marginally from 0.4% to 7.5% over the previous work [28] at a cost of providing higher assurance for the correctness of the finite precision computations of controller implementations.

6 Related Work

High-assurance control software for cyber physical systems has received much attention recently (e.g., [3,10,12,22–24,32]). Focusing on robust controller implementation, [22,32] provide simulation-based robustness analysis tools, while [3,10,12,24] studies issues related to fixed-point controller design. [4] presents a theorem proving method to verify the control related properties of Simulink models.

Moreover, there also has been work focusing on the code-level verification of controller implementation. [23,31] propose methods to check a Simulink diagram and the generated code based on the structure of the diagram and the

code, instead of input-output equivalence checking. [14,18,35,36] apply the concept of proof-carrying code to control software verification. Their approach is to annotate the code based on Lyapunov function, and prove the properties using the PVS linear algebra library [18]. However, they only consider stability and convergence properties rather than the equivalence between controller specifications and the implementations. Moreover, their verification approach may not be applicable to the code generated by existing off-the-shelf code generators because it requires the internal control of the code generators. Our own earlier work [26,28] presents methods to verify controller implementations against mathematical models, yet ignores the rounding errors in the finite precision computations of controller software implementations. There has been static analysis techniques (e.g., [5,13,16]) developed for the analysis of finite precision numerical programs, but they focus on verifying properties such as numerical stability, the absence of buffer overflow and the absence of arithmetic exception rather than verifying the equivalence between code and a dynamical system model as the specification of the controller. Finally, there has been software verification work using the model extraction technique [8,19,20,29,34], and the floating-point roundoff error estimation has been studied in [9,11,33].

7 Conclusion

We have presented an approach for the verification of finite precision implementations of linear controllers against mathematical specifications. We have proposed the use of a combination of techniques such as symbolic execution and floating point error analysis in order to extract the quantized controller model from finite precision linear controller implementations. We have defined an approximate input-output equivalence relation between the specification model (i.e., linear time-invariant model) and the extracted model (i.e., quantized controller model), and presented a method to check the approximate equivalence relation using the convex optimization formulation. We have evaluated our approach using randomly generated controller specifications and implementations by MATLAB/Simulink/Embedded Coder. The evaluation result shows that our approach is scalable for the realistic controller size, and the computational overhead to analyze the effect of floating-point error is negligible compared to our own earlier work. Future work includes the verification of a broader class of controller implementations.

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 and Future Planning (2013K1A1A2A02078326) with DGIST.

References

1. IEEE standard for floating-point arithmetic. IEEE Std 754-2008, pp. 1–70 (2008)
2. PolyFP. https://github.com/monadius/poly_fp. Accessed 2016
3. Anta, A., Majumdar, R., Saha, I., Tabuada, P.: Automatic verification of control system implementations. In: Proceedings of 10th ACM International Conference on Embedded Software, EMSOFT 2010, pp. 9–18 (2010)
4. 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)
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. ACM SIGPLAN Not. **38**, 196–207 (2003). ACM
6. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Softw. Test. Verif. Reliab. **16**(2), 97–121 (2006)
7. Clarke, L.: A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng. **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: Proceedings of the 2000 International Conference on Software Engineering, pp. 439–448. IEEE (2000)
9. Darulova, E., Kuncak, V.: Sound compilation of reals. ACM SIGPLAN Not. **49**, 235–248 (2014). ACM
10. Darulova, E., Kuncak, V., Majumdar, R., Saha, I.: Synthesis of fixed-point programs. In: Proceedings of 11th ACM International Conference on Embedded Software, EMSOFT 2013, pp. 22:1–22:10 (2013)
11. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Trans. Math. Softw. (TOMS) **37**(1), 2 (2010)
12. Eldib, H., Wang, C.: An SMT based method for optimizing arithmetic computations in embedded software code. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **33**(11), 1611–1622 (2014)
13. Feret, J.: Static analysis of digital filters. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 33–48. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24725-8_4
14. Feron, E.: From control systems to control software. IEEE Control Syst. **30**(6), 50–71 (2010)
15. Goualard, F.: How do you compute the midpoint of an interval? ACM Trans. Math. Softw. (TOMS) **40**(2), 11 (2014)
16. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 232–247. Springer, Heidelberg (2011). doi:10.1007/978-3-642-18275-4_17
17. Grant, M., Boyd, S.: CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx
18. 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)
19. Holzmann, G.J., Smith, M.H.: Software model checking: extracting verification models from source code. Softw. Test. Verif. Reliab. **11**(2), 65–79 (2001)
20. Holzmann, G.J., Smith, M.H.: An automated verification method for distributed systems software based on model extraction. IEEE Trans. Softw. Eng. **28**(4), 364–377 (2002)

21. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
22. Majumdar, R., Saha, I., Shashidhar, K., Wang, Z.: CLSE: closed-loop symbolic execution. In: NASA Formal Methods, pp. 356–370 (2012)
23. 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)
24. Majumdar, R., Saha, I., Zamani, M.: Synthesis of minimal-error control software. In: Proceedings of 10th ACM International Conference on Embedded Software, EMSOFT 2012, pp. 123–132 (2012)
25. Mathworks: Bug Reports for Incorrect Code Generation. http://www.mathworks.com/support/bugreports/?product=ALL&release=R2015b&keyword=Incorrect+Code+Generation
26. 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)
27. Park, J.: Step function example. http://dx.doi.org/10.5281/zenodo.44338
28. Park, J., Pajic, M., Lee, I., Sokolsky, O.: Scalable verification of linear controller software. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 662–679. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49674-9_43
29. Pichler, J.: Specification extraction by symbolic execution. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 462–466. IEEE (2013)
30. Rugh, W.J.: Linear System Theory. Prentice Hall, Upper Saddle River (1996)
31. Ryabtsev, M., Strichman, O.: Translation validation: from Simulink to C. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 696–701. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02658-4_57
32. Sangiovanni-Vincentelli, A., Di Natale, M.: Embedded system design for automotive applications. IEEE Comput. **10**, 42–51 (2007)
33. Solovyev, A., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 532–550. Springer, Heidelberg (2015). doi:10.1007/978-3-319-19249-9_33
34. Wang, S., Dwarakanathan, S., Sokolsky, O., Lee, I.: High-level model extraction via symbolic execution. Technical reports (CIS) paper 967, University of Pennsylvania, (2012). http://repository.upenn.edu/cis_reports/967
35. 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 (2013). arXiv preprint: arXiv:1307.2641
36. 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)
37. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, pp. 281–292. Springer, Heidelberg (2005). doi:10.1007/11408901_21