Automatic Verification of Linear Controller Software

Miroslav Pajic
Department of Electrical and Computer Engineering
Duke University
miroslav.pajic@duke.edu

Junkil Park
Department of Computer and Information Science
University of Pennsylvania
junkil.park@cis.upenn.edu

Insup Lee
Department of Computer and Information Science
University of Pennsylvania
lee@cis.upenn.edu

George J. Pappas
Department of Electrical and Systems Engineering
University of Pennsylvania
pappasg@seas.upenn.edu

Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
sokolsky@cis.upenn.edu

ABSTRACT

We consider the problem of verification of software implementations of linear time-invariant controllers. Commonly, different implementations use different representations of the controller's state, for example due to optimizations in a third-party code generator. To accommodate this variation, we exploit input-output controller specification captured by the controller's transfer function and show how to automatically verify correctness of C code controller implementations using a Frama-C/Why3/Z3 toolchain. Scalability of the approach is evaluated using randomly generated controller specifications of realistic size.

1. INTRODUCTION

Many safety- and life-critical embedded and cyber-physical systems have a software-based controller at their core. Correct operation of the controller is necessary to ensure that the system successfully achieves its mission. High degree of assurance is therefore needed in the development process of control software.

Modern controllers are designed in a model-based fashion using industry-standard tools such as Simulink. Once control design is complete, software is automatically generated from the mathematical model of the controller. The goal of our work is to develop techniques for proving that the generated code is correct with respect to the mathematical model. These techniques will allow us to avoid the need to trust the code generator, which is typically a complicated software tool that has to be revised every time a new version of the modeling environment is released.

Controllers are generally specified as a function that, given the current state of the controller and a set of input sensor values, computes control output that is sent to the system actuators and the new state of the controller. We refer to this function as the state-space representation of the controller. Implemented in software, this function is known as the *step function*. The step function is called by the control system periodically, or upon arrival of new sensor data (i.e., measurements).

In this work, we consider linear time-invariant controllers, where the relationships between the controller input and current state values, and the computed control output and updated state values are both linear. Approached naively, this linear state-space representation can be directly used as the invariant of the step function. However, code generators often optimize the control state, and these optimizations are not under the developer's control. Thus, a correctly implemented control software may not satisfy the invariant derived directly from the state-space representation.

Instead, we rely on a different specification of the controller that is insensitive to the representation of control state. This representation, based on the *transfer function* of the controller, relates the current control output to the series of past control inputs. The number of past inputs needed to capture the transfer function is known as the degree of the controller. It is well known that every state-space representation of a controller can be transformed into a transfer function, and that equivalent (i.e., similar) state-space representations will have the same transfer function [26]. In this paper, we demonstrate how the generated control code can be automatically verified with respect to a given transfer function using the popular software verification framework Frama-C [9], Why3 platform [7], and the SMT solver Z3 [11].

Verification is currently performed in the domain of real numbers, disregarding numerical errors due to floating point calculations in the software. We are planning to address the floating point domain in our future work. As the first step towards the full treatment of the problem, we consider imprecise implementations of the controller and allow coefficients of its transfer function to deviate from the specification, up to a fixed bound. We show that, while these bounded-error specifications can be handled using the same tool chain as exact specifications, they yield SMT problems with a different structure, which adversely affect scalability of the solution. We then propose an alternative, equivalent specification for the controller, which we call an instantiation-based specification. We show that by slightly increasing the size of the specification, we can dramatically improve the scalability of verification.

The contributions of this paper can be summarized as follows. We present an approach to verify software implementations of linear time-invariant controllers with respect to their mathematical specification by transfer functions. We describe a tool to perform such verification in the domain of real numbers by using an SMT solver. Finally, we explore the scalability of the approach using a set of randomly generated controllers of varying sizes.

The paper is organized as follows. Sec. 2 presents preliminar-

978-1-4673-8079-9/15/$31.00

ies on linear controllers and a couple of motivational examples for the problem considered in the paper. Sec. 3 introduces invariants for linear controllers and methods for code annotation, for both exact and inexact controller implementations. In Sec. 4, we define instantiation-based invariants for linear controllers. Finally, in Sec. 5, we present the developed framework for automatic control code verification and evaluation results, before discussing related work (Sec. 6) and providing some concluding remarks (Sec. 7).

1.1 Notation and Definitions

We use $\mathbb{R}$ to denote the set of reals, while matrix $\mathbf{I}_n$ denotes the $n \times n$ identity matrix. The i^{th} element of vector $\mathbf{x}_k$ is denoted by $\mathbf{x}_{k,i}$.[1] For vector $\mathbf{x}$, we use to denote by $|\mathbf{x}|$ the vector whose elements are absolute values of the initial vector. Also, a square matrix $\mathbf{A}$ is called *nonsingular* if its determinant is not equal to zero. Finally, for discrete-time signal $\mathbf{x}_k, k \geq 0$, the z-transform is a function of a complex variable defined as $\mathbf{X}(z) = \sum_{k=0}^{\infty} \mathbf{x}_k z^{-k}$. Rational functions are functions that can be represented by an algebraic fraction where both the numerator and the denominator are polynomial functions.

2. PRELIMINARIES ON LINEAR CONTROLLERS

The role of feedback control is to apply inputs to the plant, based on measured plant outputs, to ensure the desired behavior of the closed-loop system. We consider the general type of dynamical linear time-invariant (LTI) controllers where inputs to the controller $\mathbf{u}_k \in \mathbb{R}^p$ at each time-step k are used to compute controller outputs $\mathbf{y}_k \in \mathbb{R}^m$, which provide control inputs to the plant. We assume that controller specifications (i.e., the model) are expressed using the standard controller representation in the *state-space* form:[2]

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

Here, with $k = 0$ we denote the first execution of the controller code (i.e., the *step* function); the vector $\mathbf{z}_k \in \mathbb{R}^n$ denotes the state of the controller, while n, the size of the maintained controller state, is commonly referred to as the size of the controller. Furthermore, we assume that the specified controller has minimal realization [26], which is a common assumption, and thus n is also the degree of the controller (i.e., the degree of the denominator of its characteristic polynomial).

From (1), it follows 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 $\mathbf{D} \in \mathbb{R}^{m \times m}$ and the initial controller state $\mathbf{z}_0$, which is commonly assigned to zero, can be used to fully specify the desired controller behavior. Therefore, we will denote the controller as $\Sigma(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{z}_0)$, or just $\Sigma(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ when the initial state is zero.

To ensure the desired closed-loop system performance, matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, and $\mathbf{D}$ are obtained using standard control theory design methods. The above LTI controller form is very general and can be used to specify, for example, standard state-feedback controllers when the state is being directly measured by the sensors, observer-based feedback controllers (when direct state measurements are not available), and feedback controllers based on steady state Kalman filters (i.e., when covariance/gain matrices are not being updated). It is worth noting that, depending on the used control design method, the state of the controller can often be as large as the state of the controlled system itself.

[1] Note that we use bold letters to denote matrices and vectors (i.e., non-scalars).

[2] Matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, and $\mathbf{D}$ are commonly used to represent dynamics of the controlled physical process, while, for instance, $\mathbf{A}_c, \mathbf{B}_c, \mathbf{C}_c$, and $\mathbf{D}_c$ specify the desired controller's behavior. Similarly, usually $\mathbf{u}$ and $\mathbf{y}$ denote the plant's input and output vectors while $\mathbf{u}_c$ and $\mathbf{y}_c$ denote the controller's input and output (with $\mathbf{u}_c = \mathbf{y}$ and $\mathbf{u} = \mathbf{y}_c$). However, since in this work we do not consider dynamics of the plant but rather provide methods for automatic verification of controllers' implementations, we simplify the notation and use matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, and $\mathbf{D}$ to specify the controller, and vectors $\mathbf{u}$ and $\mathbf{y}$ to denote its input and output.

2.1 Motivating Examples

In this work, we focus on verification of controller code generated from the mathematical model in (1). However, the controller code at hand might be obtained from a code generator that performs certain optimizations that potentially violate the model (i.e., specifications), while still guaranteeing the desired control functionality and the required closed-loop performance. Here, we present two examples that illustrate such scenarios.

2.1.1 A Simple Linear Integrator

Suppose that we need to design a controller whose output y_k is a scaled sum of all previous inputs to the controller (i.e., plant measurements) $u_i \in \mathbb{R}, i = 0, ..., k-1$. This can be specified as

$$y_k = \sum_{i=0}^{k-1} \alpha u_i, k \geq 1, \quad \text{and,} \quad y_0 = 0.$$

This represents the standard integrator controller and if the Simulink *Integrator block with Forward Euler integration* is utilized, the controller will be represented in the form of (1) as $\Sigma(1, \alpha, 1, 0)$, – i.e., $z_{k+1} = z_k + \alpha u_k, y_k = z_k$. On the other hand, another realization of this controller could be $\hat{\Sigma}(1, 1, \alpha, 0)$ – i.e., $z_{k+1} = z_k + u_k, y_k = \alpha z_k$, which could introduce a lower computational error when finite precision computations are taken into account [10].

Consequently, for the above controller specification, two different controllers (i.e., controller code) will be produced by different code generation tools. Still, these two controllers Σ and $\hat{\Sigma}$ will have the same input-output behavior, while maintaining scaled and unscaled sums, respectively, of the previous values for u_k.

2.1.2 Multiple-Input-Multiple-Output Control of a Batch Reactor

In a more complex example, we consider periodic control (executed every 20 ms) of an unstable batch reactor process [17], and a stabilizing fourth order discrete-time controller $\Sigma(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{0})$ specified as

$$\mathbf{z}_{k+1} = \underbrace{\begin{bmatrix} 0.942 & 0.006888 & 0.04187 & -0.02319 \\ -0.01543 & 0.7965 & -0.03386 & 0.001563 \\ -0.1537 & 0.0137 & 0.7417 & 0.2006 \\ -0.03841 & 0.05637 & -0.02116 & 0.9949 \end{bmatrix}}_{\mathbf{A}} \mathbf{z}_k + \underbrace{\begin{bmatrix} 0.0774 & -0.0103 \\ -0.0022 & 0.0227 \\ 0.0267 & 0.0398 \\ 0.0356 & 0.0001 \end{bmatrix}}_{\mathbf{B}} \mathbf{u}_k \tag{2}$$

$$\mathbf{y}_k = \underbrace{\begin{bmatrix} 0.0583 & 0.9093 & 0.3258 & 0.08721 \\ -2.464 & -0.0504 & -1.71 & 1.165 \end{bmatrix}}_{\mathbf{C}} \mathbf{z}_k \tag{3}$$

The above controller was derived directly as an observer-based state feedback controller for the feedback and observability gain

matrices presented in [24]. Note that in order to compute updates for the state $\mathbf{z}$ as specified in (2), 16+8 = 24 multiplications need to be performed in each `step` function. In the general case, for any controller with the model as in (1), $n^2 + np = n(n + p)$ multiplications are needed to update the controller's state.

On the other hand, consider the following controller $\hat{\Sigma}(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \mathbf{0})$

$$\hat{\mathbf{z}}_{k+1} = \underbrace{\begin{bmatrix} 0.7636 & 0 & 0 & 0 \\ 0 & 0.8393 & 0 & 0 \\ 0 & 0 & 0.9595 & 0 \\ 0 & 0 & 0 & 0.9127 \end{bmatrix}}_{\hat{\mathbf{A}}} \hat{\mathbf{z}}_k + \underbrace{\begin{bmatrix} -0.2867 & -0.2581 \\ -0.3964 & -0.04506 \\ -0.07256 & 0.03278 \\ 0.5478 & -0.003331 \end{bmatrix}}_{\hat{\mathbf{B}}} \mathbf{u}_k, \quad (4)$$

$$\mathbf{y}_k = \underbrace{\begin{bmatrix} -0.1318 & 0.03834 & 0.02127 & -0.01226 \\ 0.147 & 0.08209 & -0.08674 & -0.2307 \end{bmatrix}}_{\hat{\mathbf{C}}} \hat{\mathbf{z}}_k \quad (5)$$

Here, only 4+8 = 12 multiplications are needed to compute an update for the state $\hat{\mathbf{z}}$ as in (4), because the matrix $\hat{\mathbf{A}}$ is diagonal. Note that in the case when matrix $\mathbf{A}$ in (1) is diagonal, only $n + np = n(p+1)$ multiplications are needed to update $\mathbf{z}_k$ in each `step` function.

In this example, the controllers Σ and $\hat{\Sigma}$ are *similar*, meaning that there exists a non-singular matrix $\mathbf{T}$ such that: $\hat{\mathbf{A}} = \mathbf{T}^{-1}\mathbf{A}\mathbf{T}$, $\hat{\mathbf{B}} = \mathbf{T}^{-1}\mathbf{B}$ and $\hat{\mathbf{C}} = \mathbf{C}\mathbf{T}$. This effectively implies that if the same inputs $\mathbf{u}_k$ are delivered to both controllers, the evolutions of the states $\mathbf{x}_k$ and $\hat{\mathbf{x}}_k$ would satisfy that

$$\hat{\mathbf{x}}_k = \mathbf{T}^{-1}\mathbf{x}_k, k \geq 0,$$

if and only if $\hat{\mathbf{x}}_0 = \mathbf{T}^{-1}\mathbf{x}_0$ [26], which is satisfied for these controllers as they are initialized to zero. However, what is more important, the outputs of both controllers (i.e., the input signals delivered to the controlled processes) will be identical for all k. Therefore, although it does not obey the state evolution of the initial controller Σ, and thus the controller model from (2) and (3), the controller $\hat{\Sigma}$ provides the same control functionality as Σ at a significantly reduced computational cost – making it more suitable for embedded applications.

The above two examples illustrate commonly occurring situations that code generation software for embedded applications could produce more suitable (e.g., efficient) code that might deviate from the initial controller model with the form as in (1). Even if controller code violates control specifications captured by the model $\Sigma(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$, it may still be functionally correct from the input-output perspective. Consequently, there is a need to provide verification methods based on invariants that support reasoning about correctness of linear controllers without relying only on the state-space representation of the controller. That is the focus of this work.

3. DEFINING INVARIANTS FOR LINEAR CONTROLLERS

In this section, we introduce invariants for linear controllers that can be used to verify both *state and input-output conformance* of the obtained code or only *input-output conformance* of the code. By the input-output conformance we refer to the requirement that in response to provided inputs the code provides outputs equal to the outputs provided by the model in (1) for the same input signals. Additionally, by state and input-output conformance we refer to the requirement that in response to provided inputs the code fully conforms to the initial model in (1) – i.e., not only in output but also in the internal state of the controller.

Accordingly, for verification of state and input-output (IO) conformance, invariants can be directly obtained from the model in (1). On the other hand, as illustrated in the previous section, there is a need to provide a method to capture input-output (IO) only invariants for linear controllers. These invariants cannot utilize any assertions on the controller's state, because controller implementations may be equivalent from the input-output perspective and yet rely on different state representations.

3.1 Input-Output Invariants

We consider a controller defined as $\Sigma = (\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$. The controller's transfer function $\mathbf{G}(z)$, defined as $\mathbf{G}(z) = \frac{\mathbf{Y}(z)}{\mathbf{U}(z)}$ 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. For the controller Σ we have that

$$\mathbf{G}(z) = \mathbf{C}(z\mathbf{I}_n - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D}. \quad (6)$$

In general, $\mathbf{G}(z)$ is a $m \times p$ matrix with each element $\mathbf{G}_{i,j}(z)$ being a rational function of the complex variable z. To simplify the notation, unless otherwise noted, we will assume that the considered controller is a Single-Input-Single-Output (SISO) controller, meaning that the transfer function $G(z)$ is a (single, not a matrix) rational function of z. The introduced invariants can be easily extended to Multiple-Input-Multiple-Output (MIMO) controllers.

From (6), in the general case $G(z)$ takes the form

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

where n is the size of the initial controller model, and we will also refer to n as the degree of the transfer function. In addition, $\beta_0, ..., \beta_n, \alpha_1, ..., \alpha_n \in \mathbb{R}$ and can be obtained as in (6), from the parameters of the initial controller specification (1). Therefore, the transfer function is fully described by the vectors $\alpha, \beta \in \mathbb{R}^{n+1}$ that are defined as $\alpha = [1, \alpha_1, ..., \alpha_n]$ and $\beta = [\beta_0, \beta_1, ..., \beta_n]$.

From properties of the z-transforms, the above equation implies that the controller's input and output signals satisfy the following difference equation [26]

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

with $y_k = 0, k < 0$, because $\mathbf{z}_0 = 0$ and $u_k = 0$, for $k < 0$. Thus, for any controller Σ it is possible to obtain a linear invariant of the form in (8) that specifies the relationship between controller inputs and outputs. In addition, since transfer functions are invariant to similarity transformations [26], besides the controller Σ, the linear invariant in (8) is also satisfied by any controller $\hat{\Sigma}$ obtained from the initial controller model Σ using a similarity transform with a nonsingular matrix $\mathbf{T}$.

3.2 Annotating Controller Invariants in C Code

The linear conditions in (1) and (8) respectively capture the expected state and input-output, and input-output only invariants for LTI controllers. The next challenge is to find a suitable method to express them as C code annotations, compatible with existing verification tools. To achieve this goal, we exploit ANSI/ISO C Specification Language (ACSL) [6] that enables users to specify desired

properties of C code within the program's comments. ACSL is integrated in the Frama-C platform [9] that supports tools for reasoning about correctness of C code and incorporated ACSL annotations.

To illustrate the use of ACSL to capture C code invariants, as a running example we use the following $\Sigma(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{0})$ controller

$$\mathbf{A} = \begin{bmatrix} 0.8147 & 1.1534 \\ 2.6413 & 3.6411 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 3.1019 \\ 2.1432 \end{bmatrix}, \mathbf{C} = \begin{bmatrix} 1.7121 & 0.1351 \end{bmatrix} \tag{9}$$

$$G(z) = \frac{5.60030931z^{-1} - 14.233777166248z^{-2}}{1 - 4.4558z^{-1} - 0.08007125z^{-2}} \tag{10}$$

For completeness, we first introduce annotations that capture both IO and state conformance, before introducing IO only annotations.

3.2.1 Annotating Input-Output and State Invariants

To capture the input-output and state requirements for a C function, we exploit the ACSL's notion of the function contract, which is effectively a Hoare triple [20, 13] for the entire function. ACSL utilizes the keywords `requires` and `ensures` to specify the preconditions and postconditions; the verification goal is to prove that postconditions are satisfied upon return if preconditions were satisfied when the function call occurred. The precondition for the controller's `step` function is that all pointers to memory locations are valid – for example, valid pointers to state vectors and matrix coefficients if the coefficients are not directly instantiated. This requirement is supported by the predicate `valid` that is part of ACSL.

On the other hand, the specified postconditions follow directly from the linear invariants (i.e., the model) of the controller `step` function in (1). To capture them and properly annotate the code, we exploit the built-in ACSL predicate `old` that denotes the values of a variable before the code is executed. For instance, for the considered controller defined in (9), the controller code with the annotations is presented in Listing 1.

Listing 1: Verified code for the Σ controller from (9) annotated by the state and input-output invariant

```
double x[2], u, y;
/*@ requires \valid(x+(0..1));
  @ ensures x[0] == 0.8147*\old(x[0]) +
  @       1.1534*\old(x[1]) + 3.10191*\old(u);
  @ ensures x[1] == 2.6413*\old(x[0]) +
  @       3.6411*\old(x[1]) + 2.1432*\old(u);
  @ ensures y == 1.7121*\old(x[0]) +
  @       0.1351*\old(x[1]) + 0*\old(u);
*/
void step() {
  double t1, t2;
  y = 1.7121*x[0] + 0.1351*x[1];
  t1 = 0.8147*x[0] + 1.1534*x[1] + 3.1019*u;
  t2 = 2.6413*x[0] + 3.6411*x[1] + 2.1432*u;
  x[0] = t1;
  x[1] = t2;
}
```

3.2.2 Annotating Input-Output Only Invariants

Unlike the state and IO invariants, the IO only controller invariants from (8) cannot be specified using pre- and post-conditions for every execution of the step function. This is caused by the fact that constraint (8) effectively relates the last $n+1$ executions of the `step` function. Therefore, to verify IO conformance of the controller code we have to perform execution unrolling of the `step` function a certain number of times. To achieve this, we construct the function `verif_driver` that invokes the `step` function exactly $n+1$ times. It is important to note here that the number of times the code needs to be unrolled is equal to the size of the initial controller model (i.e., the degree of transfer function) increased by 1. Finally, by using a separate label for every `step` function execution, we can then exploit the built-in ACSL keyword `at` to capture the values of input and output variables at each point of time (i.e., execution of the 'unrolled' function).

ACSL supports assertions at the end of any C code block using the `assert` keyword, where `assert p` specifies that `p` has to hold in the current state (i.e., at the place where the assertion occurs) [6]. Thus, the invariant (8) can be specified as[3]

$$\begin{aligned} &\backslash*@\ \texttt{assert}\ \backslash\texttt{at(y,}k_n\texttt{)} + \alpha_1*\backslash\texttt{at(y,}k_{n-1}\texttt{)}+\ldots \\ &\quad @\ \alpha_n*\backslash\texttt{at(y,}k_0\texttt{)} == \beta_0*\backslash\texttt{at(u,}k_n\texttt{)} +\ldots \\ &\quad @\ \beta_n*\backslash\texttt{at(u,}k_0\texttt{)} \end{aligned} \tag{11}$$

For instance, for controller Σ specified as in (9), Listing 2 presents the `verif_driver` function with the corresponding annotations.

Listing 2: Annotated code for verification of the IO only conformance of the Σ controller from (9)

```
extern double input();

void verif_driver() {
  u = input();    step();
  k0:;

  u = input();    step();
  k1:;

  u = input();    step();
  k2:;

  /*@assert \at(y,k2) - 4.4558*\at(y, k1)
    @ - 0.08007125*\at(y, k0)
    @ == 5.60030931*\at(u, k1)
    @ - 14.233777166248*\at(u, k0);
    @ */
}
```

3.3 Inexact Controller Implementations

Let us revisit the example controller with the initial model defined in (9). We obtained a computationally more efficient controller $\hat{\Sigma}(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \mathbf{0})$ via a similarity transformation from the initial controller Σ; this was done in Matlab using the function `canon` for the *modal* type of decomposition, resulting in controller $\hat{\Sigma}$

$$\hat{\mathbf{A}} = \begin{bmatrix} -0.0179 & 0 \\ 0 & 4.474 \end{bmatrix}, \hat{\mathbf{B}} = \begin{bmatrix} -1.051 \\ -1.055 \end{bmatrix}, \hat{\mathbf{C}} = \begin{bmatrix} -3.037 & -2.283 \end{bmatrix} \tag{12}$$

$$\hat{G(z)} = \frac{5.600452z^{-1} - 14.2373891245z^{-2}}{1 - 4.4561z^{-1} - 0.0800846z^{-2}} \tag{13}$$

There exists a discrepancy between transfer functions $G(z)$ in (10) and $\hat{G}(z)$ in (13), which implies that that the previously introduced

[3] If the *step* function could change the input variables, we would have to introduce separate labels for inputs and outputs (instead of a single set of k_0 to k_n points). However, to simplify the notation (and since we verify that the `step` function does not modify input variables) we use a single set of labels.

input-output invariant from (8) will not be satisfied by the control code implementing controller $\hat{\Sigma}$. Although a similarity transform results in a new controller with *the same* transfer function, due to finite-precision computation of the code generator performing controller optimization (in this case Matlab), it is possible (and expected) that the transfer function of the produced controller slightly differs from the transfer function of the initial controller.

Consequently, there is a need to extend our input-output invariants for the case with imprecise specification of the transfer functions. Specifically, we extend (7) by assuming that the transfer function could take the form as

$$G(z) = \frac{\hat{\beta}_0 + \hat{\beta}_1 z^{-1} + \cdots + \hat{\beta}_n z^{-n}}{1 + \hat{\alpha}_1 z^{-1} + \cdots + \hat{\alpha}_n z^{-n}}, \tag{14}$$

such that for $i = 0, 1, ..., n$

$$\beta_i - \epsilon_\beta \leq \hat{\beta}_i \leq \beta_i + \epsilon_\beta, \quad \alpha_i - \epsilon_\alpha \leq \hat{\alpha}_i \leq \alpha_i + \epsilon_\alpha. \tag{15}$$

Here, ϵ_β and ϵ_α denote the bounds on the errors of the transfer function coefficients. We assume that these are inputs to our verification procedure; suitable error bounds that guarantee the desired control performance can be extracted using methods from robust control theory [14], which are outside of the scope of this paper.

Yet, these inaccuracies also affect the input-output controller invariants that now need to be (re)stated. We start by noting that from (14) it holds that

$$\begin{aligned} &\exists \Delta\beta_i, \Delta\alpha_i \in \mathbb{R}, i = 0, ..., n, \ \ |\Delta\beta_i| \leq \epsilon_\beta \wedge |\Delta\alpha_i| \leq \epsilon_\alpha \wedge \\ &\quad y_k = \sum_{i=0}^{n} (\beta_i + \Delta\beta_i) u_{k-i} - \sum_{i=1}^{n} (\alpha_i + \Delta\alpha_i) y_{k-i}. \end{aligned} \tag{16}$$

However, the above condition is not linear, but rather bilinear, as it contains products $\Delta\beta_i u_{k-i}$ and $\Delta\alpha_i y_{k-i}$. Hence, we introduce additional variables $\tilde{u}_{k-i} = \Delta\beta_i u_{k-i}$ and $\tilde{y}_{k-i} = \Delta\alpha_i y_{k-i}$ and restate (16) as follows

$$\begin{aligned} &\exists \tilde{u}_{k-i}, \tilde{y}_{k-i} \in \mathbb{R}, i = 0, 1, ..., n, \\ &|\tilde{u}_{k-i}| \leq \epsilon_\beta |u_{k-i}| \wedge |\tilde{y}_{k-i}| \leq \epsilon_\alpha |y_{k-i}| \wedge \\ &y_k = \sum_{i=0}^{n} (\beta_i u_{k-i} + \tilde{u}_{k-i}) - \sum_{i=1}^{n} (\alpha_i y_{k-i} + \tilde{y}_{k-i}) \end{aligned} \tag{17}$$

Since $\epsilon_\alpha \geq 0$, the condition $|\tilde{y}_i| \leq \epsilon_\alpha |u_i|$ is equivalent to

$$((-\epsilon_\alpha y_i \leq \tilde{y}_i \leq \epsilon_\alpha y_i) \wedge (y_i \geq 0)) \vee ((\epsilon_\alpha y_i \leq \tilde{y}_i \leq -\epsilon_\alpha y_i) \wedge (y_i \leq 0)).$$

A similar term can be obtained for $|\tilde{u}_i| \leq \epsilon_\beta |u_i|$. Thus, we introduce a predicate `error_bound(a,b,c)` as

```
#define error_bound(a,b,c) (((b)>=0 && -(b)*(c)
<= (a) <= (b)*(c)) || ((b)<0 && (b)*(c) <= (a)
<= -(b)*(c)))
```

With the above notation, and using the ACSL keyword `exists` for the existential quantifier, the input-output invariant (17) can be annotated in code as shown in (18). For example, for the controller Σ specified (9), Listing 3 illustrates the `verif_driver` function with the input-output invariant annotations that allow for transfer function inaccuracies.

It is important to highlight that the IO invariant in (17) and the corresponding code annotation in (18) exploit a mixture of both universal and existential quantifiers. Existential quantifiers are used to specify tolerance variables $\tilde{y}$ and $\tilde{u}$, while universal quantifiers are employed since (17) has to hold for all values of u_k at points k_0, ... k_n and y_k at k_0, ... k_{n-1} (where label k_0 does not have to correspond to any time-step k). Note that the use of formulas with both universal and existential quantifiers usually presents a challenge for SMT solvers (e.g., Z3), which, as we will illustrate in the evaluation section (Section 5.1), significantly limits scalability of the approach and degrees of controllers that can be verified using the invariant. We address this problem in the next section as we provide another approach to derive input-output invariants for LTI controllers.

Listing 3: Annotated code for verification of the IO conformance within the tolerance limit for the example controller from (9); Note that $\tilde{y}$ and $\tilde{u}$ from (18) are denoted by `yt` and `ut`

```
extern double input();

void verif_driver() {
  u = input();    step();
  k0:;

  u = input();    step();
  k1:;

  u = input();    step();
  k2:;

  /*@assert \exists real yt0, yt1, ut0, ut1;
    @ error_bound(yt0, \at(y, k0), 0.01) &&
    @ error_bound(yt1, \at(y, k1), 0.01) &&
    @ error_bound(ut0, \at(u, k0), 0.01) &&
    @ error_bound(ut1, \at(u, k1), 0.01) &&
    @ \at(y,k2)
    @ - 4.4558*\at(y, k1) + yt1
    @ - 0.08007125*\at(y, k0) + yt0
    @ == 5.60030931*\at(u, k1) + ut1
    @ - 14.233777166248*\at(u, k0) + ut0;
    @ */
}
```

4. INSTANTIATION-BASED INPUT-OUTPUT INVARIANTS FOR LTI CONTROLLERS

In this section, we present an alternative method to specify linear invariants that are equivalent to the IO invariant introduced in (8) (and (17), (18)). As we will show, the method is better suited to capture robust invariants that allow for slightly inexact controller implementations, as in cases when there exists a small discrepancy between the transfer function of the initial controller and the one implemented by the provided code.

Initially, we consider the exact input-output invariants from (8), and we start by logically 'unrolling' the condition (8) N times – by summarizing N executions of the controller from (8) using the matrices introduced below.

DEFINITION 1. *Consider controller Σ. For the controller's inputs and outputs u_k and y_k at time steps $k = 0, 1, ..., n + N - 1$, we define the matrix $\mathbf{D}_N = \begin{bmatrix} \mathbf{D}_N^y & \mathbf{D}_N^u \end{bmatrix}$ where*

$$\mathbf{D}_N^y = \begin{bmatrix} y_n & y_{n-1} & \cdots & y_1 & y_0 \\ y_{n+1} & y_n & \cdots & y_2 & y_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ y_{n+N-1} & y_{n+N-1} & \cdots & y_N & y_{N-1} \end{bmatrix} \tag{19}$$

$$\mathbf{D}_N^u = \begin{bmatrix} u_n & u_{n-1} & \cdots & u_1 & u_0 \\ u_{n+1} & u_n & \cdots & u_2 & u_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ u_{n+N-1} & u_{n+N-2} & \cdots & u_N & u_{N-1} \end{bmatrix} \tag{20}$$

```
\*@ assert \exists real ỹ_0, ..., ỹ_{n-1}, ũ_0, ..., ũ_n
  @ error_bound(ỹ_0, \at(y, k_0), ε_α) && ... && error_bound(ỹ_{n-1}, \at(y, k_{n-1}), ε_α) &&
  @ error_bound(ũ_0, \at(u, k_0), ε_β) && ... && error_bound(ũ_n, \at(u, k_n), ε_β) &&
  @ (\at(y, k_n)+α_1*\at(y, k_{n-1})+ỹ_{n-1}+...+α_n*\at(y, k_0)+ỹ_0 == β_0*\at(u, k_n)+ũ_n+...+β_n*\at(u, k_0)+ũ_0)
```
(18)

Consequently, from (8) and the above definition it follows that

$$\mathbf{D}_N \cdot \theta = \mathbf{0}, \tag{21}$$

where $\theta = \begin{bmatrix}1 & \alpha_1 & ... & \alpha_n & \beta_0 & \beta_1 & ... & \beta_n\end{bmatrix}^T = \begin{bmatrix}\alpha^T & \beta^T\end{bmatrix}^T$ captures all of the parameters of the controller's transfer function.

The following proposition shows that under certain conditions, linear equalities from (21) are equivalent to the invariant in (8) obtained from the controller's transfer function.

PROPOSITION 1. *Consider LTI controller Σ of size n. Then the rank of any matrix $\mathbf{D}_N$ cannot be larger than $2n+1$. Furthermore, when the rank of $\mathbf{D}_N$ is $2n+1$, then linear conditions from* (21) *are satisfied if and only if the condition* (8) *is satisfied for all k.*

PROOF. From Definition 1, $\text{rank}(\mathbf{D}_N) \leq 2n+2$, for any $N \geq 1$, because the matrix has $2n+2$ columns. Note that the matrix cannot have rank $2n+2$ as that would imply that the columns of $\mathbf{D}_N$ are linearly independent and thus their linear combination $\mathbf{D}_N^y \cdot \theta$ could be equal to the zero vector only if all elements of θ are zero (i.e., $\theta = \mathbf{0}$). This is clearly not possible since 1 is the first element of θ.

Now suppose that $\text{rank}(\mathbf{D}_N) = 2n+1$. As we argued before, from (8) and Definition 1 we have that (21) is satisfied. Thus, let's consider the other direction.

We start by assuming that (21) holds for a vector θ obtained from some vectors α and β. Since Σ is an LTI controller of size n then, as presented in Section 3, there exist vectors $\hat{\alpha}, \hat{\beta}$, and $\hat{\theta} = \begin{bmatrix}\hat{\alpha}^T & \hat{\beta}^T\end{bmatrix}^T$ for which (8) is satisfied for each k. Therefore, since $\mathbf{D}_N$ captures inputs and outputs of the system (from its definition), we have that

$$\mathbf{D}_N \cdot \hat{\theta} = \mathbf{0} = \mathbf{D}_N \cdot \theta \Rightarrow \mathbf{D}_N \cdot (\theta - \hat{\theta}) = \mathbf{0}. \tag{22}$$

Note that since the first element of $\theta - \hat{\theta}$ is zero, $\mathbf{D}_N \cdot (\theta - \hat{\theta})$ presents linear combination of all columns of $\mathbf{D}_N$ except the first one. Thus, from (22), if $\theta \neq \hat{\theta}$ it follows that the remaining $2n+1$ columns of $\mathbf{D}_N$ (i.e., without the first column) are linearly dependent. On the other hand, the first column of $\mathbf{D}_N$ presents a linear combination of other columns with coefficients from $\hat{\theta}$. Thus, since the rank of $\mathbf{D}_N$ is $2n+1$, we have that the remaining $2n+1$ columns are linearly independent, which contradicts are previous conclusion. Thus, we have that $\theta = \hat{\theta}$, meaning that if (21) holds so does (8), which concludes the proof. □

The specific structure of matrix $\mathbf{D}_N$ (the matrices with structure such as $\mathbf{D}_N^y$ and $\mathbf{D}_N^u$ are called *Toeplitz matrices*) makes it suitable to obtain the rank of $\mathbf{D}_N$ equal to $2n+1$ with exactly $N = 2n+1$ rows. To generate matrix $\mathbf{D}_{2n+1}$ with rank $2n+1$, we start by assigning $y_k = 0$ and $u_k = 0$ for all $k = 0, ..., n-1$, and then $u_n = 1$. After this, the only assignments are done on $u_k, k > n$, as the values for $y_k, k > n$ are derived from the initial controller model (i.e., specification). Specifically, after assigning $u_n = 1$, we set the next $n-1$ inputs to zero. Since n is the size of the initial controller (which is minimal by our assumption), the corresponding first n rows of both $\mathbf{D}^y$ and $\mathbf{D}^u$ will be linearly independent. Finally, the last $n+1$ inputs $u_k, k = 2n, ..., 3n$, are assigned in a way that ensures that each newly introduced row is linearly independent of the previous ones – this is easy to achieve due to the fact that inputs $u_k, k = n+1, ..., 2n-1$ were all zero.

The above proposition allows us to specify a set of $2n+1$ linear invariants, which if satisfied would verify input-output conformance of the considered controller code – i.e., the invariant in (8). At first glance, the benefits of using the invariant with $2n+1$ linear conditions might be unclear, when an invariant with a single linear condition can be used. However, as we discussed at the end of the previous section, the invariant in (8) and its corresponding ACSL annotation (11) require that for all values of u at points k_0 ... k_n and y at k_0 ... k_{n-1}, the value of y at k_n is equal to the specified linear combination of u_k's and y_k's. On the other hand, the invariant (21) does not use the universal quantifier; rather, it specifies that if values of u_k at $3n+1$ points are equal to the corresponding values from $\mathbf{D}_{2n+1}^u$ and the values of y_k at the first n points are equal to the corresponding values from $\mathbf{D}_{2n+1}^y$, then the values of y_k at the remaining $2n+1$ points have to be equal to the remaining values from the matrix $\mathbf{D}_{2n+1}^y$.

Finally, the above method for deriving a set of linear invariants exploits a similar approach as the ones used in testing for system identification. By creating a suitable matrix $\mathbf{D}_{2n+1}$ we effectively provide a set of controller inputs at consecutive executions of the `step` function and verify whether the controller outputs conform to the prespecified input-output behavior of the controller. Hence, we refer to the linear invariants specified in (21) as *instantiation-based invariants*.

4.1 Defining Instantiation-Based Invariants as Code Annotation

Similarly to the IO controller invariants from (8), to introduce instantiation-based invariants as code annotations we have to perform execution unrolling of the `step` function within a newly defined `verif_driver` function. Due to the fact that the matrix $\mathbf{D}_{2n+1}$ contains controller inputs and outputs for steps 0 to $3n$, we need to unroll the function exactly $3n+1$ times and introduce a separate label $k_i, i = 0, ..., 3n$ for each `step` function execution, as previously presented in Listing 3. With this notation, the invariant from (21) can be captured as the code annotation from (23), where u_i and $y_i, i = 0, 1, ..., 3n$, specify the corresponding elements of the matrix $\mathbf{D}_{2n+1}$ as stated in Definition 1.

Another approach to define instantiation-based invariants is to directly perform input variables assignments in `verif_driver` code, as presented in Listing 4. This effectively reduces the complexity of the assert statement, whose form is shown in (24). In Section 5.1, we will compare efficiency of these approaches.

REMARK 1. *The above annotations can be significantly simplified if we know the variables in the code used to maintain the controller's state (for example, this can be determined with the use of static analysis tools). As previously described, the matrix $\mathbf{D}_{2n+1}$ is designed in a way that $u_k = 0$ and $y_k = 0$ for $k = 0, ..., n-1$. For linear systems with minimal realizations (which means that they are controllable and observable [26]) this would also imply that the state of the controller at time $n-1$ would have to be zero (i.e., $\mathbf{z}_{n-1} = 0$). Thus, in this case, we would need to unroll code*

$$\backslash*@\ \texttt{assert}\ ((\backslash\texttt{at(y,}k_0\texttt{)}==y_0)\ \&\&\ \ldots\ \&\&\ (\backslash\texttt{at(y,}k_{n-1}\texttt{)}==y_{n-1})\ \ \&\&\ \ (\backslash\texttt{at(u,}k_0\texttt{)}==u_0\ )\ \&\&\ \ldots\ \&\&\ (\backslash\texttt{at(u,}k_{3n}\texttt{)}==u_{3n}))\ @\Rightarrow\ ((\backslash\texttt{at(y,}k_n\texttt{)}\ ==\ y_n)\ \ \&\&\ \ \ldots\ \ \ \&\&\ \ (\backslash\texttt{at(y,}k_{3n}\texttt{)}==y_{3n}))\tag{23}$$

$$\backslash*@\ \texttt{assert}\ ((\backslash\texttt{at(y,}k_0\texttt{)}==y_0)\ \&\&\ \ldots\ \&\&\ (\backslash\texttt{at(y,}k_{n-1}\texttt{)}==y_{n-1}))\ \Rightarrow\ ((\backslash\texttt{at(y,}k_n\texttt{)}==y_n)\ \&\&\ \ldots\ \&\&\ (\backslash\texttt{at(y,}k_{3n}\texttt{)}==y_{3n}))\tag{24}$$

execution only 2n+1 times (and introduce only $2n+1$ *points/labels) by either specifying* $\mathbf{z}_{n-1} == 0$ *as part of the* `assert` *statement similar to what is done in* (23)*, or introduce an additional assignment* $\mathbf{z}_{n-1} = 0$ *in the* `verify_driver` *function with an* `assert` *statement similar to the one in* (24)*.*

Listing 4: One structure of the code annotations for verification of the IO only conformance using the *Instantiation-based Invariants* from (21); Note that u_t denotes u_t from the matrix $\mathbf{D}_{2n+1}$

```
extern double input();

void verif_driver()  {
  u = u_0;    step();
  k0:;

  u = u_1;    step();
  k1:;
  .
  .
  u = u_3n;    step();
  k3n:;

  /*@assert ... ( from (24) )    @*/
}
```

4.2 Instantiation-Based Invariants for Inexact Controller Implementations

The invariant introduced in (21) can be especially important for verification of inexact controller implementations that allow for small errors in the coefficients of the implemented controllers' transfer functions. To elaborate on this, let's use the same notation as in Section 3.3 and let's assume that transfer function can be specified using the vector $\hat{\theta} = \begin{bmatrix}\hat{\alpha}^T & \hat{\beta}^T\end{bmatrix}^T$, where $\hat{\alpha}, \hat{\beta}$ satisfy (15). Thus, from (21) we have that $\mathbf{D}_{2n+1}\cdot\hat{\theta} = \mathbf{0}$ which is (as in Proposition 1) equivalent to the invariant in (16).

Now, by introducing $\Delta\theta = \hat{\theta} - \theta$, we have that

$$\mathbf{D}_{2n+1}\cdot\theta + \mathbf{D}_{2n+1}\cdot\Delta\theta = \mathbf{0}. \tag{25}$$

Since the matrix $\mathbf{D}_{2n+1}$ and the initial transfer function vectors α and β are known, from the initial controller model, we can compute

$$\mathbf{v} = -\mathbf{D}^y_{2n+1}\alpha - \mathbf{D}^u_{2n+1}\beta.$$

Using the vector $\mathbf{v}$, we can state the following invariant

$$\begin{aligned}\exists\Delta\beta_i, \Delta\alpha_i \in \mathbb{R}, i = 0, ..., n, |\Delta\beta_i| \le \epsilon_\beta \wedge |\Delta\alpha_i| \le \epsilon_\alpha \wedge \\ \mathbf{D}^y_{2n+1}\Delta\alpha + \mathbf{D}^u_{2n+1}\Delta\beta = \mathbf{v}\ \wedge \\ y_{n+i} = \mathbf{D}^y_{2n+1}(n+i),\ \ i = 0, ..., 2n,\end{aligned} \tag{26}$$

where $D^y_{2n+1}(k)$ denotes the entry in the matrix D^y_{2n+1} on the position corresponding to y_k as defined in (19) (for the exact controller specification). The above invariant is linear and utilizes only the existential quantifier. Again, as in the case for the exact IO invariant, we can define two types of instantiation-based invariants for inexact controller implementations. For instance, the assert statement similar to the one in (23), for exact controller implementations, is introduced in (27). Here, a and b are used to represent $\Delta\alpha_i$ and $\Delta\beta_i$, and we introduced a predicate `vector_equal(x,y)` that compares vectors $\mathbf{x}$, $\mathbf{y}$ and operator `lin_comb(D,a1,...,an)` that presents the linear combination of n columns of `D` with weights `a1,...,an`.

5. FRAMEWORK FOR AUTOMATIC VERIFICATION

In this section, we present the developed automatic verification framework based on the previously described invariants for LTI controllers (see Fig. 1). To automatically verify C code annotated with ACSL specification [6], we employ the popular software verification platform Frama-C [9]. We also exploit WP [5], a plugin of Frama-C that enables deductive verification of C code with ACSL annotations. Given annotated C code, Frama-C/WP parses the code and performs the weakest precondition calculations to analyze the validity of the annotations in the code. For each annotation, Frama-C/WP generate a set of proof obligations to establish that the C code satisfies the annotated specification.

Frama-C/WP supports generation of proof obligations in the intermediate specification language WhyML [1]. The generated proof obligations in WhyML can be submitted to various theorem provers via the Why3 platform [7], both automatic theorem provers (e.g., Z3 [11]) or interactive theorem provers (e.g., Coq [4]). To automate the verification process, we employ the automatic theorem prover Z3 to discharge the proof obligations. Z3 is an SMT solver that checks satisfiability of a given formula modulo a certain theory, and to check the validity of the proof goal of a proof obligation, we used Why3 to generate an SMT instance for Z3.

While transforming annotated C code to an SMT instance along the toolchain in Fig. 1, we observed that WP and Why3 tend to generate the declarations for some extra theories in their outputs; these are not necessary to prove the proof goal, but could adversely affect the performance of the SMT solving with Z3. In addition, some of the generated declarations in the intermediate specifications are not directly relevant to the proof goal, while others are redundant since they have been already incorporated in Z3. Therefore, to improve the performance at the SMT solving stage, we created an automated Python script to intervene in the transformation and remove unnecessary theory declarations from the intermediate specifications such as the proof obligations in WhyML and SMT instances.

In the deductive verification of the `verif_driver` function, which as described in Sections 3 and 4 by construction invokes the `step` function a certain number of times, the function contract (i.e., pre- and post-condition) of the `step` function would be required for the deduction rule for the function calls. However, it is very difficult to specify the function contract of the `step` function without knowing its input-output and state invariant (and which we in the general case do not know). Thus, to avoid writing the `step` function specification, we preprocess the code performing the function inlining for the `step` function (i.e., inserting the body of the `step` function wherever the function is called in the code). More-

```
\*@ assert \exists real a_0,...,a_{n-1},b_0,...,b_n
  @ (a_0 ≤ ε_α) && (a_0 ≥ −ε_α) && ... && (a_{n-1} ≤ ε_α) && (a_{n-1} ≥ −ε_α) && (b_0 ≤ ε_β) && (b_0 ≥ −ε_β) && ... && (b_n ≤ ε_β) && (b_n ≥ −ε_β)
\*@  ((\at(y,k_0)==y_0)  && ... &&  (\at(y,k_{n-1})==y_{n-1})  &&  (\at(u,k_0)  ==u_0  )  && ... &&  (\at(u,k_{3n})==u_{3n}))
  @ ⇒  ((\at(y,k_n)  ==  y_n)  &&  ...  &&  (\at(y,k_{3n})==y_{3n})  &&
\*@  vector_equal((lin_comb(Dy,1,a_0,...,a_{n-1})  +  lin_comb(Du,b_0,...,b_n)),v)  )
```

(27)

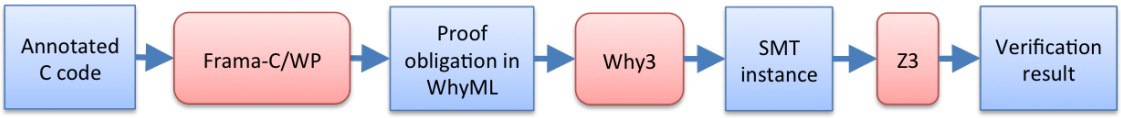

Figure 1: The verification toolchain.

over, the `step` function may contain loops. Note that it is challenging to automate the deductive verification of C code with loops when no loop invariants are provided. To avoid synthesizing the invariants of the loops in the `step` function, we transform the code by unrolling the loops in it. This is possible when the loops have some constant upper bounds. We note that the size of the controller for embedded system is statically fixed in many cases, and the upper bound of the loops are normally bounded in terms of the size of the controller.

Finally, Frama-C/WP supports two different models for floating-point arithmetic operations of C code: float model and real model. In the float model, when deriving the weakest precondition WP performs floating-point operations as defined in the IEEE 754 floating-point standard. This results in generated proof obligations that are too complex to be handled by existing automatic theorem provers. On the other hand, the real model transforms floating-point operations to operations on reals, thus enabling the SMT solvers that support arithmetic theory of reals to discharge the generated proof obligations. As previously stated, in this work we employ the real model, considering the problem of the bounded error specifications as the first step toward the full treatment of the problem. Addressing floating-point computations is an avenue for future work.

5.1 Evaluation

To evaluate the developed verification framework, we first considered the controller specification (i.e., model) from (9). We first verified that the `step` function from Listing 1 satisfies the state and IO invariants for the $\Sigma(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{0})$ model (9). In addition, we verified that the controller Σ satisfies the IO only invariants annotated in the `verif_driver` function from Listing 2. Using the IO invariants that allow for inexact controller implementations, as specified in the `verif_driver` function from Listing 3, we verified the correctness of the `step` function implementing computationally more efficient controller $\tilde{\Sigma}$ from (12), with transfer function (13). Finally, we exploited both types of `assert` statements from (23) and (24) to specify instantiation based invariants; we verified IO conformance of the example controller Σ from (9) and the inexact controller $\tilde{\Sigma}$ from (12) using the invariant (27).

Furthermore, we verified randomly generated controllers of varying size and analyzed how different types of the introduced invariants affect scalability of the verification approach. We also illustrated the use of the developed framework on verification of LTI controllers automatically generated by Simulink Coder from both discrete-time *State-Space* and *LTI System* library blocks. Note that, although these blocks can be used to specify the same mathematical model, the structures of the actual code generated from these blocks are significantly different.

We evaluated verification performance for both 'exact' and 'inexact' input-output invariants. We considered five different types of invariants: (a) IO and state invariants (denoted by SS invariants) introduced in (1); (b) IO only invariants based on the transfer function (denoted by TF), defined in (8) and (11), (c) Instantiation-based IO invariants defined in (21) and (23) (referred to as IB'_{3n+1}), (d) Instantiation-based IO invariants defined in (21) and (24) (referred to as IB''_{3n+1}), (e) Instantiation based IO invariants for only $2n+1$ points when the state variable is known, as described in Remark 1 on (23) – referred to as IB'_{2n+1}, (f) Instantiation-based IO invariants for only $2n+1$ points when the state variable is known and based, as described in Remark 1, on (24) – referred to as IB'_{2n+1}. Except the SS invariants, all other types of invariants were evaluated for both exact and inexact controller implementations.

Fig. 2 presents measured Z3 running times for verification of random controller implementations that exactly implement specified transfer functions. Note that for each considered controller size n, we randomly generated 50 controllers. Fig. 2 presents the average running times (along with the ranges of running times) for different controller size n and different type of invariants. As expected, the use of SS invariants scales best. However, note that SS invariants can be used **only** when we know the implemented state-space model of the controller. On the other hand, the use of TF invariants also scales well when the exact transfer function of the implemented controller is known. Finally, due to the size of the generated proof obligations for both IB'_{3n+1} and IB''_{3n+1} invariants, verification using these invariants takes the most time.

To evaluate verification performance with inexact controller implementations, for each of the different controller sizes n we generated 50 random controller models. Then, for each model we would try to verify an implementation of a controller similar to the initial controller (i.e., with the same transfer function), in order to obtain controllers with inexact implementations. Since we could not know the state invariants for these controllers, we were not able to test the use of SS invariants. The results of our experiments are presented in Fig. 3. Our first observation is that with inexact implementations, the TF-invariants based verification scales very poorly; we were not able to verify the TF invariants for controllers with more than two states. The reason for this is that TF invariants employ both universal and existential quantifiers, as we have discussed in Section 3. On the other hand, as expected (due to the use of only existential quantifiers) verification of instantiation-based invariants scales reasonably well (both IB'_{3n+1} and IB''_{3n+1}).

6. RELATED WORK

The problem of providing high-assurance software for embedded control systems and cyber-physical systems has recently attracted

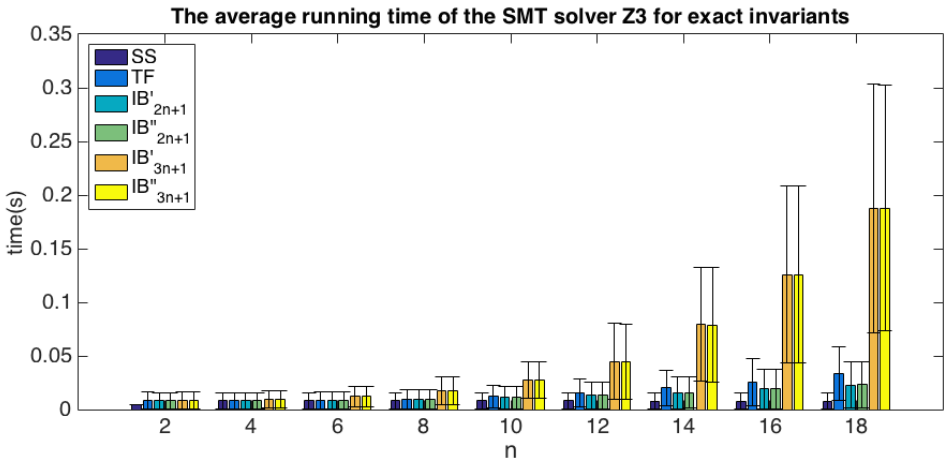

Figure 2: Z3 running times for LTI controller verification using five different types of controller invariants.

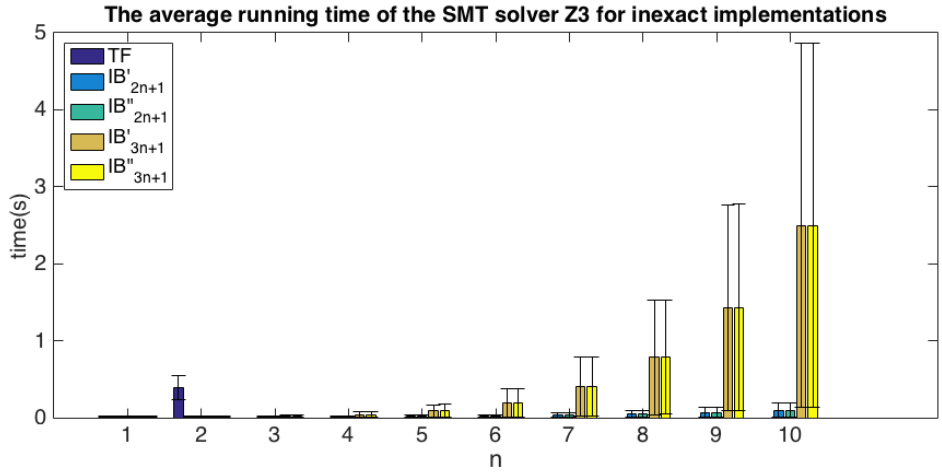

Figure 3: Z3 running times for verification of LTI controllers using 'inexact' invariants for all five different types of controller invariants. Note that in this case, verification of TF invariants does not scale well because controllers with the size greater than two can not be verified.

significant attention (e.g., [27, 2, 24, 23, 22, 10, 15]). One line of work has focused on robust implementations of embedded controllers. For instance, in [27] the authors present a model-based simulation platform that can be used to analyze controller robustness against different implementation issues, including sampling, quantization, and fixed-point arithmetic. [2, 24] present methods for design of robust fixed-point controllers that guarantee stability and minimize implementation errors, respectively. In [22], the authors introduce a robustness analysis tool that computes the maximum deviation of the plant states due to measurement uncertainties. The use of SMT solvers for synthesis of fixed-point embedded software has been addressed in [10, 15].

On the other hand, the problem of verification of control code has received less attention. The work in [23] introduces equivalence checking between hierarchical Simulink models and generated controller code. However, only the structure of generated code is considered, and its compliance with the structure of the initial Simulink model is checked – without taking into account implementation of the code blocks/functions. The authors in [3] present a method for verification of Simulink models by translating them to Why3 [7] models. Yet, the verification is again performed only on the model level and not on the code level.

A closely related work was on the concept of proof-carrying code for control software [16, 19, 29, 28]. Still, the authors introduce code annotations, based on Lyapunov functions, that capture control-related properties of the controller that guarantee only closed-loop system stability [16], and that cannot as in our work be used to check for correctness of the controller implementation. In [29, 28], autocoding that annotates code with these stability invariants is presented. Also, in [19], the authors introduce PVS linear algebra libraries that can be used for verification of stability of a closed-loop system controlled by a software implementation in C.

Finally, Frama-C [9] and ACSL [6] have been widely used for software verification. For example, for verification of a subset of the standard C library [8], safety-critical software in the railway domain [18], and the Xen kernel [25]). In addition, [12, 21] present methods for dynamic analysis in Frama-C, and in [19] the authors present the use of Frama-C for verification of control software.

7. DISCUSSION AND CONCLUSION

We have presented an approach to verify the generated controller code against the mathematical model used for controller design. This allows us to obtain a higher degree of assurance for the control code by removing the need to trust a code generator. We have proposed to use invariants based on transfer functions, a well-known concept in the linear systems theory, since it allows us to accom-

modate optimizations in the state representation that could be applied by the code generator. We have demonstrated the feasibility of performing automatic verification of such invariants on controllers with realistic number of states. We have studied both exact and inexact controller implementations; the latter may result from numerical manipulations within the code generator. For inexact implementations, the invariant incorporates error bounds on the level of deviation from the transfer function. We evaluated our approach on controller implementations, generated by Matlab for randomly generated transfer functions. The evaluation also showed that scalability of verification can be improved by using an alternative representation of the transfer function.

An important avenue of future work is to incorporate into the analysis the effects of numerical errors resulting from floating point calculations in the control code. We believe that our work on inexact controller implementations can be extended to cover numerical errors as well.

Acknowledgments

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 also supported in part by Global Research Laboratory Program (2013K1A1A2A02078326) through NRF, and the DGIST Research and Development Program (CPS Global Center) funded by the Ministry of Science, ICT & Future Planning.

8. REFERENCES

[1] The WhyML Programming Language, http://why3.lri.fr/doc-0.80/manual004.html.

[2] A. Anta, R. Majumdar, I. Saha, and P. Tabuada. Automatic verification of control system implementations. In *Proc. 10th ACM International Conference on Embedded Software*, EMSOFT'10, pages 9–18, 2010.

[3] D. Araiza-Illan, K. Eder, and A. Richards. Formal verification of control systems' properties with theorem proving. In *UKACC International Conference on Control (CONTROL)*, pages 244–249, 2014.

[4] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, et al. The Coq proof assistant reference manual: Version 6.1. 1997.

[5] P. Baudin, F. Bobot, L. Correnson, and Z. Dargaye. WP 0.8 manual - Frama-C. Technical report, CEA LIST, 2014.

[6] P. Baudin, P. Cuoq, J.-C. Filliatre, C. Marche, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C Specification Language, Version 1.4. Technical report, CEA LIST and INRIA, 2010.

[7] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[8] N. Carvalho, C. da Silva Sousa, J. S. Pinto, and A. Tomb. Formal Verification of kLIBC with the WP Frama-C Plug-in. In *NASA Formal Methods*, pages 343–358. Springer, 2014.

[9] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c. In *Software Engineering and Formal Methods*, pages 233–247. 2012.

[10] E. Darulova, V. Kuncak, R. Majumdar, and I. Saha. Synthesis of fixed-point programs. In *Proc. 11th ACM International Conference on Embedded Software*, EMSOFT'13, pages 22:1–22:10, 2013.

[11] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. 2008.

[12] M. Delahaye, N. Kosmatov, and J. Signoles. Common specification language for static and dynamic analysis of C programs. In *Proc. 28th Annual ACM Symposium on Applied Computing*, pages 1230–1235, 2013.

[13] E. W. Dijkstra. *A discipline of programming*. Prentice-Hall Englewood Cliffs, 1976.

[14] G. Dullerud and F. Paganini. *Course in Robust Control Theory*. Springer-Verlag New York, 2000.

[15] H. Eldib and C. Wang. 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.

[16] E. Feron. From control systems to control software. *Control Systems, IEEE*, 30(6):50–71, 2010.

[17] M. Green and D. J. Limebeer. *Linear robust control*. Courier Corporation, 2012.

[18] K. Hartig, J. Gerlach, J. Soto, and J. Busse. Formal Specification and Automated Verification of Safety-Critical Requirements of a Railway Vehicle with Frama-C/Jessie. In *FORMS/FORMAT 2010*, pages 145–153. 2011.

[19] H. Herencia-Zapana, R. Jobredeaux, S. Owre, P.-L. Garoche, E. Feron, G. Perez, and P. Ascariz. PVS linear algebra libraries for verification of control software algorithms in C/ACSL. In *NASA Formal Methods*, pages 147–161. 2012.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10):576–580, 1969.

[21] N. Kosmatov and J. Signoles. A lesson on runtime assertion checking with Frama-C. In *Runtime Verification*, pages 386–399, 2013.

[22] R. Majumdar, I. Saha, K. Shashidhar, and Z. Wang. CLSE: Closed-loop symbolic execution. In *NASA Formal Methods*, pages 356–370. 2012.

[23] R. Majumdar, I. Saha, K. Ueda, and H. Yazarel. Compositional equivalence checking for models and code of control systems. In *52nd Annual IEEE Conference on Decision and Control (CDC)*, pages 1564–1571, 2013.

[24] R. Majumdar, I. Saha, and M. Zamani. Synthesis of minimal-error control software. In *Proc. 10th ACM International Conference on Embedded Software*, EMSOFT'12, pages 123–132, 2012.

[25] A. Puccetti. Static Analysis of the XEN Kernel using Frama-C. *Journal of Universal Computer Science*, 16(4):543–553, 2010.

[26] W. J. Rugh. *Linear system theory*. Prentice Hall, 1996.

[27] A. Sangiovanni-Vincentelli and M. Di Natale. Embedded system design for automotive applications. *IEEE Computer*, (10):42–51, 2007.

[28] T. Wang, R. Jobredeaux, H. Herencia, P.-L. Garoche, A. Dieumegard, E. Feron, and M. Pantel. From design to implementation: an automated, credible autocoding chain for control systems. *arXiv preprint arXiv:1307.2641*, 2013.

[29] T. E. Wang, A. E. Ashari, R. J. Jobredeaux, and E. M. Feron. Credible autocoding of fault detection observers. In *American Control Conference (ACC)*, pages 672–677, 2014.