2018

Automatic Verification Of Linear Controller Software

Junkil Park
University of Pennsylvania, park11@seas.upenn.edu

Automatic Verification Of Linear Controller Software

Abstract
Many safety-critical cyber-physical systems have a software-based controller at their core. Since the system behavior relies on the operation of the controller, it is imperative to ensure the correctness of the controller to have a high assurance for such systems. Nowadays, controllers are developed in a model-based fashion. Controller models are designed, and their performances are analyzed first at the model level. Once the control design is complete, software implementation is automatically generated from the mathematical model of the controller by a code generator.

To assure the correctness of the controller implementation, it is necessary to check that the code generation is correctly done. Commercial code generators are complex black-box software that are generally not formally verified. Subtle bugs have been found in commercially available code generators that consequently generate incorrect code. In the absence of verified code generators, it is desirable to verify instances of implementations against their original models. Such verification is desired to be performed from the input-output perspective because correct implementations may have different state representations to each other for several possible reasons (e.g., code generator's choice of state representation, optimization used in code generator and code transformation).

In this dissertation, we propose several methods to verify a given controller implementation against its given model from the input-output perspective. First of all, we propose a method to derive assertions from the controller model, and check if the assertions are invariant to the controller implementation via a proposed toolchain based on a popular deductive program verification framework. Moreover, we propose an alternative more scalable method that extracts a model from the controller implementation using the symbolic execution technique, and compare the extracted model to the original controller model using state-of-the-art constraint solvers. Lastly, we extend our latter method to correctly account for the rounding errors in the floating-point computation of the controller implementation. We demonstrate the scalability of our proposed approaches through evaluation with randomly generated controller specifications of realistic size.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Insup Lee

Second Advisor
Oleg Sokolsky

Keywords
Controller software verification, Cyber-Physical Systems, Model-based development

Subject Categories
Computer Engineering | Computer Sciences

AUTOMATIC VERIFICATION OF LINEAR CONTROLLER SOFTWARE

Junkil Park

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2018

Soli Deo gloria

Acknowledgements

First, I would like to thank and acknowledge my advisor, Prof. Insup Lee. Through his generous support, expert guidance and unwavering encouragement, I was able to complete my doctoral research and this dissertation. He always encouraged me to collaborate with my colleagues and helped me to grow as an independent and proactive researcher. I am also thankful for his helping and guiding me to find my research topic and direction and help to write good research papers.

I am also deeply thankful to Prof. Oleg Sokolsky, my co-advisor, for helping and supporting me throughout my Ph.D. studies. I am thankful for our many discussions and the good advice and ideas that he gave. His door was always open and he always listened and counseled me.

I would also like to thank the members of my committee, Prof. Rajeev Alur, my committee chair, Prof. Mayur Naik, Prof. James Weimer, and Prof. Miroslav Pajic for their insightful feedback for improving my dissertation. I am thankful for the time that they gave for my proposal and defense and for their support to help me finish.

I would like to say a special thanks to Prof. Miroslav Pajic. During his years at PRECISE lab at Penn, he worked with me and helped me to find my research topic and worked together to write and publish several papers.

Without his continual guidance, discussions and help with my research, I would not have been able to complete this dissertation.

I would like to mention those who I had the pleasure to work with and learned from in PRECISE lab. I want to express my deep thanks to Prof. Nicola Bezzo, Dr. Radoslav Ivanov, Wenrui Meng, Sangdon Park and many others. I am also thankful to have been able to discuss online and receive much help from Dr. Nicky Williams, Dr. Alexey Solovyev, Prof. Matthieu Martel and Dr. Nasrine Damouche.

I want to thank my church, Philadelphia UBF, for their prayer support during my Ph.D. studies. They gave me strength and encouragement when things were difficult. I am thankful for their spiritual direction and prayers that I received that helped me to finish this marathon.

Personally, I would like to thank my family who was there for me throughout all my years of my Ph.D. studies. I want to thank my parents, Kiyoung Park and Youngnim Kim, and my sister, Jinsin Park, for their constant support and love. I want to thank my in-laws, Dr. Henry and Pauline Park, and John and Helen Ross, who were my family in America and supported me and my family whenever we needed help.

Last but not least, I want to thank my precious family, my wife, Pauline, and my kids, Joshua and Jenna, who were both born during my Ph.D. studies and brought me increasing joy and motivation throughout my past years. I am sincerely thankful for Pauline, who is my best friend and most suitable helper. Accompanying me through anxious days and sleepless nights, her loving care and unwavering support made this journey possible. I am deeply indebted to her, and I dedicate this dissertation to her.

ABSTRACT

AUTOMATIC VERIFICATION OF LINEAR CONTROLLER SOFTWARE

Junkil Park

Insup Lee

Oleg Sokolsky

Many safety-critical cyber-physical systems have a software-based controller at their core. Since the system behavior relies on the operation of the controller, it is imperative to ensure the correctness of the controller to have a high assurance for such systems. Nowadays, controllers are developed in a model-based fashion. Controller models are designed, and their performances are analyzed first at the model level. Once the control design is complete, software implementation is automatically generated from the mathematical model of the controller by a code generator.

To assure the correctness of the controller implementation, it is necessary to check that the code generation is correctly done. Commercial code generators are complex black-box software that are generally not formally verified. Subtle bugs have been found in commercially available code generators that consequently generate incorrect code. In the absence of verified code generators, it is desirable to verify instances of implementations against their original models. Such verification is desired to be performed from the input-output perspective because correct implementations may have different state representations to each other for several possible reasons (e.g., code generator's choice of state representation, optimization used in code generator and code transformation).

In this dissertation, we propose several methods to verify a given controller implementation against its given model from the input-output perspective. First of all, we propose a method to derive assertions from the controller model, and check if the assertions are invariant to the controller implementation via a proposed toolchain based on a popular deductive program verification framework. Moreover, we propose an alternative more scalable method that extracts a model from the controller implementation using the symbolic execution technique, and compare the extracted model to the original controller model using state-of-the-art constraint solvers. Lastly, we extend our latter method to correctly account for the rounding errors in the floating-point computation of the controller implementation. We demonstrate the scalability of our proposed approaches through evaluation with randomly generated controller specifications of realistic size.

Contents

List of Tables

List of Illustrations

Chapter 1

Introduction

1.1 Motivation

Most safety- and life-critical embedded and cyber-physical systems have a software-based controller at their core. The safety of these systems rely on the correct operation of the controller. Thus, in order to have a high assurance for such systems, it is imperative to ensure that controller software is correctly implemented.

Nowadays, controller software is developed in a model-based fashion, using industry-standard tools such as Simulink [73] and Stateflow [77]. In this development process, first of all, the controller model is designed and analyzed. Controller design is performed using a mathematical model of the control system that captures both the dynamics of the “plant”, the entity to be controlled, and the controller itself. With this model, analysis is performed to conclude whether the plant model adequately describes the system to be controlled, and whether the controller achieves the desired goals of the control system. Once the control engineer is satisfied with the design, a software implementation is automatically produced by code generation from the mathematical model of the controller. Code generation tools such as Embedded Coder [72] and Simulink Coder [74] are widely used. The generated controller implementation is either used as it is in the control system, or sometimes transformed into another code before used for various reasons such as numerical accuracy improvement [23, 24] and code protection [18, 15, 11]. For simplicity's sake, we will call code generation even when code generation is potentially followed by code transformation.

To assure the correctness of the controller implementation, it is necessary to check that code generation is correctly done. Ideally, we would like to have verified tools for code generation. In this case, no verification of the controller implementation would be needed because the tools would guarantee that any produced controller correctly implements its model. In practice, however, commercial code generators are complex black-box software that are generally not amenable to formal verification. Subtle bugs have been found in commercially available code generators that consequently generate incorrect code [71]. Unverified code transformers may introduce unintended bugs in the output code.

In the absence of verified code generators, it is desirable to verify instances of implementations against their original models. Therefore, the goal of this work is to develop an automatic method to perform such instance verification for a given controller model and software implementation.

1.2 Problem Formulation

This work considers the problem of verifying software implementations of controllers against controller models as mathematical specifications. We assume that control design activities have been performed, achieving the acceptable degree of assurance for the control design. Thus, 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.

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. In this work, we focus on linear-time invariant (LTI) controllers, since these are the most commonly used controllers in control systems. In LTI controllers, the relationships between the controller input and current state values, and the computed control output and updated state values are both linear.

In software, controllers are implemented as a subroutine (or a function in the C language). This function is known as the *step function.* The step function is invoked by the control system periodically, or upon arrival of new sensor data (i.e., measurements).

To properly address this verification problem, the following challenges should be considered: First of all, such verification should be performed from the input-output perspective (i.e., input-output conformance). Correct implementations may have different state representations to each other for several possible reasons (e.g., code generator's choice of state representation, optimization used in the code generation process). In other words, the original controller model and a correct implementation of the model may be different from each other in state representation, while being functionally equivalent from the input-output perspective. Thus, it is necessary to develop the verification technique that is not sensitive to the state representation of the controller.

Moreover, there is an inherent discrepancy between controller models and their implementations. The controller software for embedded systems uses a finite precision arithmetic (e.g., floating-point arithmetic) which introduces rounding errors in the computation. The effect of these rounding errors needs to be considered in the verification process. In addition to these rounding errors, the implementations may be inexact in the numeric representation of controller parameters due to the potential rounding errors in the code generation/optimization process. Thus, it is reasonable to allow a tolerance in the conformance verification as long as the implementation has the same desired property to the model's.

Finally, such verification is desired to be automatic and scalable because verification needs to be followed by each instance of code generation. In the next section, we describe the contributions of our proposed methods that address this problem.

1.3 Contributions

At a high level, the goal of this work is to ensure the correctness of controller implementation instances with respect to their original models in the ab-

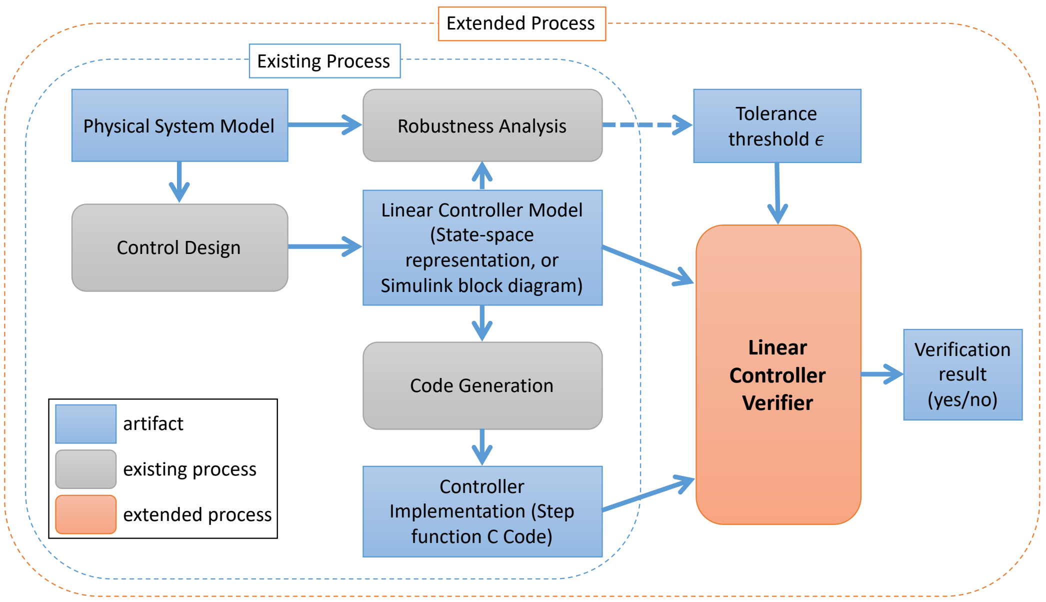
Extended Process
Existing Process
Physical System Model
Robustness Analysis
Tolerance threshold ϵ
Control Design
Linear Controller Model (State-space representation, or Simulink block diagram)
Linear Controller Verifier
Verification result (yes/no)
Code Generation
Controller Implementation (Step function C Code)
artifact
existing process
extended process

Figure 1.1: Our extended proposed process for linear controller verification

sence of verified code generators. Thus, as shown in Figure 1.1, we propose an extended process building upon the existing model-based development process. The main new entity in the extended proposed process is Linear Controller Verifier (LCV), an automatic tool to verify a step function C code (i.e., controller implementation) against an LTI controller model from the input-output perspective with tolerance up to a given threshold value ϵ.[1] To develop LCV, we explore two alternative approaches in this dissertation: one is based on invariant checking while another is based on similarity checking. The more specific contributions that this dissertation make are as follows:

First of all, we propose an invariant checking-based verification method [58]. Given a controller model, this method derives assertions to be satisfied by

[1]We assume that a threshold value ϵ is given by a control engineer as a result of robustness analysis that guarantees the desired properties of the control system in the presence of uncertain disturbances.

correct step functions. These assertions exactly capture the specification of the controller, thus the problem of verifying step function is reduced to the problem of checking whether these assertions are invariant to the step function or not. These assertions enable the verification of the input-output only conformance, because they are stated over the input and output variables only, and no state variables appear in the assertions. In order to do this, 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. Moreover, given a tolerance threshold by a control engineer, we provide a way to relax the invariants (i.e., assertions) of the controller code in order to tolerate inexact controller parameters up to the threshold. We demonstrate how the generated control code can be automatically verified with respect to a given transfer function using the popular deductive software verification framework Frama-C [22], Why3 platform [13], and the SMT solver Z3 [26].

Secondly, we propose a similarity checking-based verification method [60]. This approach is based on extracting a model from the controller code and establishing equivalence between the original and the extracted models. This similarity checking-based method significantly improves the scalability of verification compared to the invariant checking-based method. The main reason is that the similarity checking-based method symbolically executes the given controller code only one time, thus avoiding the loop/execution unrolling that the invariant checking-based method involves. The first step of the similarity checking-based verification is to extract a model from the given controller code using the symbolic execution technique. The symbolic expressions identified as the result of symbolic execution are used to reconstruct the model of the controller. Next, the reconstructed model is checked for input-output equivalence against the given original model, using the well-known necessary and sufficient condition for the equivalence of two minimal LTI models. We account for the numerical errors of the inexact controller parameters by allowing for a bounded discrepancy between the models in the equivalence checking. We provide two ways to automatically check the equivalence based on an SMT problem formulation and a convex optimization formulation respectively.

Thirdly, building on the work of the similarity checking-based method, we propose an extended verification approach that correctly accounts for the floating-point calculation of controller implementation. In this extended method, we newly introduce error terms into the representation of the extracted model that characterize the effects of floating-point rounding errors. We use an optimization formulation to perform approximate equivalence checking. We demonstrate that this extended approach suffers only minimal degradation in performance while offering a higher assurance of the floating-point controller implementation.

Lastly, we develop LCV, the prototype tool in Figure 1.1 that implements our verification approaches. The tool accepts a subset of Simulink block diagrams (i.e., LTI) as input and performs conformance checking against the given implementations. We demonstrate that the tool are able to detect some known reproduced bugs of the code generator Embedded Coder [72], and an unknown bug of the code optimizer Salsa [24].

1.4 Related Work

This section provides a brief summary of related work, and argues the reason why the current techniques are insufficient in coping with the proposed problem in this thesis. To ensure the correctness of controller implementation against the controller model, a typically used method in practice is equivalence testing (or back-to-back testing) [70, 19, 20] which compares the outputs of executable model and code for the common input sequence. The limitation of this testing-based method is that it does not provide a thorough verification.

Static analysis-based approaches [12, 34, 39] have been used to analyze the finite-precision numerical controller code, but focuses on checking common properties such as numerical stability, the absence of buffer overflow or arithmetic exceptions rather than verifying the code against the model.

The work of [65, 52] proposes translation validation techniques for Simulink diagrams and the generated codes. The verification relies on the structure of the block diagram and the code, thus being sensitive to the controller state while our method verifies code against the model from the input-output perspective, not being sensitive to the controller state. Due to optimization and transformation during a code generation process, a generated code which is correct may have different state representation than the models. In this case, our method can verify that the code is correct with respect to the model, but the state-sensitive methods [65, 52] cannot.

[35, 42, 81, 80] present a control software verification approach based on the concept of proof-carrying code. In their approach, the code annotation based on the Lyapunov function and its proof are produced at the time of code generation. The annotation asserts control theory related properties such as stability and convergence, but not equivalence between controller specifications and the implementations. In addition, their approach requires the control of code generator, and may not be applicable to the off-the-shelf black-box code generators.

There is a line of work that has focused on robust implementations of embedded controllers. For instance, in [66] 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. [5, 53] present methods for design of robust fixed-point controllers that guarantee stability and minimize implementation errors, respectively. In [51], 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 [25, 33].

Firnally, the authors in [6] present a method for verification of Simulink models by translating them to Why3 [13] models. Yet, the verification is again performed only on the model level and not on the code level.

1.5 Outline of the Dissertation

The rest of this dissertation is organized as follows:

Chapter 2 provides a background of this dissertation including LTI systems with motivating examples, and an overview of software verification techniques used in this work.

Chapter 3 describes an invariant checking-based approach to verify software implementations of LTI controllers with respect to their mathematical specifications by transfer functions. This chapter describe a toolchain developed to perform such verification, and demonstrate the scalability of the approach using a set of randomly generated controllers of varying sizes.

Chapter 4 presents a similarity checking-based approach to verify controller implementations by extracting models from the implementations and comparing the extracted models against the original models. This chapter also demonstrate the scalability of the prototype tool of this approach.

Chapter 5 presents a method that extends the similarity checking-based approach of Chapter 4. This extended method correctly accounts for the rounding errors that would occur in the floating-point computation of the controller implementation. We demonstrate the scalability of our proposed approaches through evaluation with randomly generated controller specifications of realistic size.

Chapter 6 describes LCV, the prototype tool that implements our verification approaches. This chapter also evaluates the tool LCV through the case study and the scalability analysis.

Chapter 7 concludes the dissertation and discuss future research opportunities.

Chapter 2

Preliminaries

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

2.1 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.

A discrete system takes a discrete-time signal $\mathbf{u}_k$, $k \geq 0$, as input and generates a discrete-time signal $\mathbf{y}_k$ as output in response to the input. The system may have a hidden internal state. In this case, the output signal $\mathbf{y}_k$

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

is influenced by not only the input signal $\mathbf{u}_k$ but also the internal state of the system at time k. The change of the internal state is influenced by the input signal $\mathbf{u}_k$ and the current internal state. A discrete system is said to be *linear* if $\alpha\mathbf{y}_k + \beta\hat{\mathbf{y}}_k$ is the output of the system in response to the input $\alpha\mathbf{u}_k + \beta\hat{\mathbf{u}}_k$ for any scalars α and β when $\mathbf{y}_k$ and $\hat{\mathbf{y}}_k$ are the outputs of the systems in response to the input $\mathbf{u}_k$ and $\hat{\mathbf{u}}_k$ respectively. Moreover, a system is said to be *time-invariant* if $\mathbf{y}_{k-k_0}$ is the output of the system in response to the input $\mathbf{u}_{k-k_0}$ for any k_0 when $\mathbf{y}_k$ is the output of the system in response to the input $\mathbf{u}_k$.

Finally, for discrete-time signal $\mathbf{x}_k, k \geq 0$, the z-transform is a function of a complex variable defined as $\mathbf{X}(z) = \sum_{k=0}^{\infty} \mathbf{x}_k z^{-k}$. For the signal $\mathbf{x}_k$ in the time domain, this z-transform produces a new presentation $\mathbf{X}(z)$ in the *z-domain*. The z-transform is considered as the discrete analogue of the Laplace transform [64]. Rational functions are functions that can be represented by an algebraic fraction where both the numerator and the denominator are polynomial functions.

2.2 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 thesis, 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:

$$\begin{aligned}\mathbf{z}_{k+1} &= \mathbf{A}\mathbf{z}_k + \mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{z}_k + \mathbf{D}\mathbf{u}_k.\end{aligned} \tag{2.1}$$

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 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 thesis, we assume that data is exchanged with the step function through global variables.[2] In other words, the input, output and state variables are declared in the global scope, and the step function

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

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.

2.3 Motivating Examples

To motivate our work, we introduce two examples, which illustrate limitations of the standard verification techniques that directly utilize the mathematical model from (6.1), in cases when controller software is generated by a code generator whose optimizations potentially violate the model while still ensuring the desired control functionality.

2.3.1 A Scalar Linear Integrator

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

$$y_k = \sum_{i=0}^{k-1} \alpha u_i, k > 1, \quad \text{and,} \quad y_0 = 0. \tag{2.2}$$

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., $z_{k+1} = z_k + \alpha u_k, y_k = z_k$. On the other hand, a realization $\hat{\Sigma}(1, 1, \alpha, 0)$ – i.e., $z_{k+1} = z_k + u_k, y_k = \alpha z_k$, of the controller would introduce a reduced computational error when finite precision arithmetics is used [25]. Thus, controller specification (3.7) 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.

2.3.2 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

$$\mathbf{z}_{k+1} = \underbrace{\begin{bmatrix} -0.500311 & 0.16751 & 0.028029 & -0.395599 & -0.652079 \\ 0.850942 & 0.181639 & -0.29276 & 0.481277 & 0.638183 \\ -0.458583 & -0.002389 & -0.154281 & -0.578708 & -0.769495 \\ 1.01855 & 0.638926 & -0.668256 & -0.258506 & 0.119959 \\ 0.100383 & -0.432501 & 0.122727 & 0.82634 & 0.892296 \end{bmatrix}}_{\mathbf{A}} \mathbf{z}_k + \\ + \underbrace{\begin{bmatrix} 1.1149 & 0.164423 \\ -1.56592 & 0.634384 \\ 1.04856 & -0.196914 \\ 1.96066 & 3.11571 \\ -3.02046 & -1.96087 \end{bmatrix}}_{\mathbf{B}} \mathbf{u}_k \tag{2.3}$$

$$\mathbf{y}_k = \underbrace{\begin{bmatrix} 0.283441 & 0.032612 & -0.75658 & 0.085468 & 0.161088 \\ -0.528786 & 0.050734 & -0.681773 & -0.432334 & -1.17988 \end{bmatrix}}_{\mathbf{C}} \mathbf{z}_k \tag{2.4}$$

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.

$$\hat{\mathbf{z}}_{k+1} = \underbrace{\begin{bmatrix} 0.87224 & 0 & 0 & 0 & 0 \\ 0 & 0.366378 & 0 & 0 & 0 \\ 0 & 0 & -0.540795 & 0 & 0 \\ 0 & 0 & 0 & -0.332664 & 0 \\ 0 & 0 & 0 & 0 & -0.204322 \end{bmatrix}}_{\hat{\mathbf{A}}} \hat{\mathbf{z}}_k + $$

$$+ \underbrace{\begin{bmatrix} 0.822174 & -0.438008 \\ -0.278536 & -0.824313 \\ 0.874484 & 0.858857 \\ -0.117628 & -0.506362 \\ -0.955459 & -0.622498 \end{bmatrix}}_{\hat{\mathbf{B}}} \mathbf{u}_k, \tag{2.5}$$

$$\mathbf{y}_k = \underbrace{\begin{bmatrix} -0.793176 & 0.154365 & -0.377883 & -0.360608 & -0.142123 \\ 0.503767 & -0.573538 & 0.170245 & -0.583312 & -0.56603 \end{bmatrix}}_{\hat{\mathbf{C}}} \hat{\mathbf{z}}_k \tag{2.6}$$

The above controllers Σ and $\hat{\Sigma}$ are *similar*, 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 (6.1), would require $n^2 + np = n(n + p)$ multiplications to update its state. This can be significantly reduced when matrix $\mathbf{A}$ in (6.1) is diagonal – in this case only $n + np = n(p + 1)$ multiplications are needed.

2.4 Software Verification Techniques

This section briefly overviews the software verification techniques such as deductive verification, symbolic execution and model extraction.

2.4.1 Deductive Verification

Deductive verification [36] is a deductive approach to verify a program, which normally consists of two steps: (1) turning the correctness of a program into a mathematical statement (also known as verification condition), and then (2) proving the statement. The correctness of a program is defined by a specification of the program. A specification can be given using the concept of *Hoare triple* [43]. A Hoare triple is the form $\{P\}s\{Q\}$ where P is a precondition, s is a program statement, and Q is a postcondition. A program statement s is said to be *correct* with respect to some given precondition P and postcondition Q when the Hoare triple $\{P\}s\{Q\}$ is valid. The Hoare triple $\{P\}s\{Q\}$ is valid if the execution of s starting from any state that satisfies P finishes in a state that satisfies Q. Note that if s is not terminating, it is correct for any P and Q. In this regard, the validity of a Hoare triple asserts the *partial correctness* of a program. An additional requirement for s to be terminating defines *total correctness*. One can establish the validity of a Hoare triple using the Hoare rules with providing proper intermediate assertions. However, this typically requires much manual effort in practice. Thus, most modern verification condition generators use the weakest precondition calculation which computes the weakest precondition $wp(s, Q)$ for some given program statement s and postcondition Q such that $\{wp(s, Q)\}s\{Q\}$. Consequently, the validity of the Hoare triple $\{P\}s\{Q\}$ is equivalent to $P \implies wp(s, Q)$. Generated verification conditions can be discharged by the support of tools such as SMT solvers (e.g., Z3 [26], CVC4 [8]) and interactive theorem provers (e.g., Coq [7], PVS [57]), possibly being coordinated by a multi-prover deductive verification framework [37].

Finally, Frama-C [22] and ACSL [10] have been widely used for software verification. For example, for verification of a subset of the standard C library [16], safety-critical software in the railway domain [41], and the Xen kernel [63]). In addition, [28, 48] present methods for dynamic analysis in Frama-C, and in [42] the authors present the use of Frama-C for verification of control software.

2.4.2 Symbolic Execution

Symbolic execution [47] [17] is a program analysis technique which executes a program in a symbolic manner to explore multiple different execution paths.

Symbolic execution contrasts with concrete execution. Concrete execution for program analysis can be said to be program testing, which process is as follows: concrete values are given as an input to a program under test. The program is executed for the concrete inputs, and the observable behavior (e.g., output) of the execution is inspected to see if it is expected or not. In this process, a concrete execution yields a single execution path. In most cases, concrete executions only cover a small subset of the whole input space, and thus may miss the program executions which actually lead to errors.

On the other hand, symbolic execution allows a program to take as input symbolic values instead of concrete values. A symbolic value (e.g., α, β) denotes an arbitrary concrete value. For a given symbolic input, a symbolic execution engine explores the control flow paths of the program while maintaining (1) a symbolic program state which maps variables to symbolic expressions, and (2) a path condition which is a constraint on the symbolic input values and characterizes the path currently being explored. In other words, a path condition is the conjunction of the conditions of the branches taken along the path. During a symbolic execution, branch statements update the path condition, while assignment statements update the symbolic program state.

Symbolic execution can be used in program analysis in many different ways. First of all, symbolic execution can be used to generate test cases that covers certain execution paths. Suppose that there is an execution path with path condition C. The feasibility of the path is reduced to the satisfiability of the path condition C. Constraint solvers (e.g., Z3 [26] and CVC4 [8]) are used to automatically find an assignment which satisfies C, which can serve as a concrete input (i.e., test case) that covers the path. Moreover, symbolic execution can be used to verify assertions in programs. Suppose that the path with the path condition C reaches an assertion statement which asserts the predicate P. If $(C \implies P)$ is valid, it is guaranteed that all concrete input values that lead to the path (i.e., satisfies C) are not violating the assertion P. Checking the validity of the formula can also be done automatically by constraint solvers. If the formula is not valid, the solvers also provide a concrete assignment (i.e., concrete input value) that causes the assertion violation.

2.4.3 Model Extraction

The model extraction technique has been used in software verification [78, 21, 45, 46, 54, 79, 62, 67]. There is a line of work that has focused on using the model extraction technique for software model checking [78, 21, 45, 46, 54]. From a given source program, these model extraction tools automatically extract a verification model in the input language of several existing model checkers such as SPIN [44], SMV [55], SAL [27] and Zing [4]. Bandera [21] takes Java programs, and extracts models from the programs in a certain intermediate representation which are further translated into the input languages of existing model checkers such as SPIN, SMV and SAL. Modex [45, 46] extract the control-flow skeleton of a given C program in the Promela language [44] to verify the message passing operations of the program using the SPIN model checker. The work [54] focuses on extracting verification models from C programs of Windows kernel drivers to facilitate software model checking using the Zing model checker.

There is also much work on extracting high-level state machine models from source programs [49, 69, 79, 67]. These approaches reconstruct a state machine model from a given program for the use of program testing and code review (i.e., visualizing the state machine model for a programmer to understand the high-level perspective of the behavior of the legacy program). For model extraction, the symbolic execution technique is used in [49, 79, 67] while the work [69] analyze the structure of the abstract syntax tree of the program.

Finally, the work [62] applies the symbolic execution technique to implemented source code to extract mathematical functional models. The approach only considers a restricted set of programs that can be represented as pure mathematical functions (i.e., without having states), thus being unable to account for persistent static variables such as global variables which are essential to represent the state of controllers in controller implementations.

Chapter 3

Invariant Checking-based Verification Approach

This chapter describe an invariant checking-based verification method [58]. Given a controller model, this method derives assertions to be satisfied by correct step functions. These assertions exactly capture the specification of the controller, thus the step function verification problem is reduced to the problem of checking whether these assertions are invariant to the step function or not. In order to derive invariants that assert the input-output only conformance of code against model, 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 [64]. In this chapter, 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 [22], Why3 platform [13], and the SMT solver Z3 [26].

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.

This chapter is organized as follows: Section 3.1 introduces invariants for linear controllers and methods for code annotation, for both exact and inexact controller implementations. In Sec. 3.2, we define instantiation-based invariants for linear controllers. Finally, in Sec. 3.3, we present the developed framework for automatic control code verification and evaluation results.

3.1 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 (6.1) 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 (6.1) – 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 (6.1). 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.

3.1.1 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

$$\mathbf{G}(z) = \mathbf{C}(z\mathbf{I}_n - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D}. \tag{3.1}$$

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

$$G(z) = \frac{\beta_0 + \beta_1 z^{-1} + \cdots + \beta_n z^{-n}}{1 + \alpha_1 z^{-1} + \cdots + \alpha_n z^{-n}}, \tag{3.2}$$

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 (3.1), from the parameters of the initial controller specification (6.1). 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 the properties of the z-transforms such as linearity and time-invariance, the above equation implies that the controller's input and output signals satisfy the following difference equation [64]:

$$y_k = \sum_{i=0}^{n} \beta_i u_{k-i} - \sum_{i=1}^{n} \alpha_i y_{k-i}, \tag{3.3}$$

with $y_k = 0, k < 0$, because $\mathbf{z}_0 = 0$ and $u_k = 0$, for $k < 0$. The coefficients α and β of this equation come from (3.2). Thus, for any controller Σ it is possible to obtain a linear invariant of the form in (3.3) that specifies the relationship between controller inputs and outputs. In addition, since transfer functions are invariant to similarity transformations [64], besides the controller Σ, the linear invariant in (3.3) is also satisfied by any controller $\hat{\Sigma}$ obtained from the initial controller model Σ using a similarity transform with a nonsingular matrix $\mathbf{T}$.

3.1.2 Annotating Controller Invariants in C Code

The linear conditions in (6.1) and (3.3) 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) [10] that enables users to specify desired properties of C code within the program's comments. ACSL is integrated in the Frama-C platform [22] 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

$$\mathbf{A} = \begin{bmatrix} 0.8147 & 1.1534 \\ 2.6413 & 3.6411 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 3.1019 \\ 2.1432 \end{bmatrix}, \mathbf{C} = \begin{bmatrix} 1.7121 & 0.1351 \end{bmatrix} \tag{3.4}$$

$$G(z) = \frac{5.60030931z^{-1} - 14.233777166248z^{-2}}{1 - 4.4558z^{-1} - 0.08007125z^{-2}} \tag{3.5}$$

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

3.1.3 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 [43, 30] 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 (6.1). 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 (3.4), the controller code with the annotations is presented in Listing 1.

3.1.4 Annotating Input-Output Only Invariants

Unlike the state and IO invariants, the IO only controller invariants from (3.3) cannot be specified using pre- and post-conditions for every execution of the step function. This is caused by the fact that constraint (3.3) effectively

```
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;
}
```

Listing 1: Verified code for the Σ controller from (3.4) annotated by the state and input-output invariant

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) [10]. Thus, the invariant (3.3) can be specified as[1]

$$\begin{aligned}&\texttt{/*@ assert \textbackslash at(y,}k_n\texttt{) + }\alpha_1\texttt{*\textbackslash at(y,}k_{n-1}\texttt{)+...}\\ &\quad\texttt{@ }\alpha_n\texttt{*\textbackslash at(y,}k_0\texttt{) == }\beta_0\texttt{*\textbackslash at(u,}k_n\texttt{) +...}\\ &\quad\texttt{@ }\beta_n\texttt{*\textbackslash at(u,}k_0\texttt{) */}\end{aligned} \tag{3.6}$$

For instance, for controller Σ specified as in (3.4), Listing 2 presents the `verif_driver` function with the corresponding annotations.

[1]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.

```
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);
    @ */
}
```

Listing 2: Annotated code for verification of the IO only conformance of the Σ controller from (3.4)

3.1.5 Inexact Controller Implementations

Let us revisit the example controller with the initial model defined in (3.4). 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 $\hat{\Sigma}$:

$$\hat{\mathbf{A}} = \begin{bmatrix} -0.0179 & 0 \\ 0 & 4.474 \end{bmatrix}, \hat{\mathbf{B}} = \begin{bmatrix} -1.051 \\ -1.055 \end{bmatrix}, \hat{\mathbf{C}} = \begin{bmatrix} -3.037 & -2.283 \end{bmatrix} \tag{3.7}$$

$$\hat{G(z)} = \frac{5.600452z^{-1} - 14.2373891245z^{-2}}{1 - 4.4561z^{-1} - 0.0800846z^{-2}} \tag{3.8}$$

There exists a discrepancy between transfer functions $G(z)$ in (3.5) and $\hat{G}(z)$ in (3.8), which implies that that the previously introduced input-output invariant from (3.3) 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 (3.2) by assuming that the transfer function could take the form as

$$G(z) = \frac{\hat{\beta}_0 + \hat{\beta}_1 z^{-1} + \cdots + \hat{\beta}_n z^{-n}}{1 + \hat{\alpha}_1 z^{-1} + \cdots + \hat{\alpha}_n z^{-n}}, \tag{3.9}$$

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

$$\beta_i - \epsilon_\beta \leq \hat{\beta}_i \leq \beta_i + \epsilon_\beta, \qquad \alpha_i - \epsilon_\alpha \leq \hat{\alpha}_i \leq \alpha_i + \epsilon_\alpha. \tag{3.10}$$

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 [31].

Yet, these inaccuracies also affect the input-output controller invariants that now need to be (re)stated. We start by noting that from (3.9) it holds that

$$\begin{aligned} \exists \Delta\beta_i, \Delta\alpha_i \in \mathbb{R}, i = 0, ..., n, \quad & |\Delta\beta_i| \leq \epsilon_\beta \wedge |\Delta\alpha_i| \leq \epsilon_\alpha \wedge \\ y_k = \sum_{i=0}^{n} (\beta_i + \Delta\beta_i) u_{k-i} - & \sum_{i=1}^{n} (\alpha_i + \Delta\alpha_i) y_{k-i}. \end{aligned} \tag{3.11}$$

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 (3.11) as follows

$$\begin{aligned} & \exists \tilde{u}_{k-i}, \tilde{y}_{k-i} \in \mathbb{R}, i = 0, 1, ..., n, \\ & |\tilde{u}_{k-i}| \leq \epsilon_\beta |u_{k-i}| \wedge |\tilde{y}_{k-i}| \leq \epsilon_\alpha |y_{k-i}| \wedge \\ y_k = & \sum_{i=0}^{n} (\beta_i u_{k-i} + \tilde{u}_{k-i}) - \sum_{i=1}^{n} (\alpha_i y_{k-i} + \tilde{y}_{k-i}) \end{aligned} \tag{3.12}$$

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 (3.12) can be annotated in code as follows:

```
/*@ assert \exists real $\tilde{y}_0, ..., \tilde{y}_{n-1}, \tilde{u}_0, ..., \tilde{u}_n$
  @ error_bound($\tilde{y}_0$,\at(y,$k_0$),$\epsilon_\alpha$) && ...  &&
  @ error_bound($\tilde{y}_{n-1}$,\at(y,$k_{n-1}$),$\epsilon_\alpha$) &&
  @ error_bound($\tilde{u}_0$,\at(u,$k_0$),$\epsilon_\beta$) && ...  &&          (3.13)
  @ error_bound($\tilde{u}_n$,\at(u,$k_n$),$\epsilon_\beta$) &&
  @ (\at(y,$k_n$)+$\alpha_1$*\at(y,$k_{n-1}$)+$\tilde{y}_{n-1}$+...+$\alpha_n$*\at(y,$k_0$)+$\tilde{y}_0$
  @ == $\beta_0$*\at(u,$k_n$)+$\tilde{u}_n$+...+$\beta_n$*\at(u,$k_0$)+$\tilde{u}_0$) */
```

For example, for the controller Σ specified (3.4), 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 (3.12) and the corresponding code annotation in (3.13) 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 (3.12) 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 later (Section 3.3.1), 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.

3.2 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 (3.3), (3.12) and (3.13). 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 (3.3), and we start by logically 'unrolling' the condition (3.3) N times – by summarizing N executions of the controller from (3.3) using the matrices introduced below.

Definition. *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* $\mathbf{D}_N =$

```
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;
    @ */
}
```

Listing 3: Annotated code for verification of the IO conformance within the tolerance limit for the example controller from (3.4); Note that $\tilde{y}$ and $\tilde{u}$ from (3.13) are denoted by `yt` and `ut`

$\begin{bmatrix}\mathbf{D}_N^y & \mathbf{D}_N^u\end{bmatrix}$ *where*

$$\mathbf{D}_N^y = \begin{bmatrix} y_n & y_{n-1} & \dots & y_1 & y_0 \\ y_{n+1} & y_n & \dots & y_2 & y_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ y_{n+N-1} & y_{n+N-1} & \dots & y_N & y_{N-1} \end{bmatrix} \tag{3.14}$$

$$\mathbf{D}_N^u = \begin{bmatrix} u_n & u_{n-1} & \dots & u_1 & u_0 \\ u_{n+1} & u_n & \dots & u_2 & u_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ u_{n+N-1} & u_{n+N-2} & \dots & u_N & u_{N-1} \end{bmatrix} \tag{3.15}$$

Consequently, from (3.3) and the above definition it follows that

$$\mathbf{D}_N \cdot \theta = \mathbf{0}, \tag{3.16}$$

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

The following proposition shows that under certain conditions, linear equalities from (3.16) are equivalent to the invariant in (3.3) 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* (3.16) *are satisfied if and only if the condition* (3.3) *is satisfied for all k.*

Proof. From Definition 3.2, 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 (3.3) and Definition 3.2 we have that (3.16) is satisfied. Thus, let's consider the other direction.

We start by assuming that (3.16) 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 3.1, there exist vectors $\hat{\alpha}, \hat{\beta}$, and $\hat{\theta} = \begin{bmatrix} \hat{\alpha}^T & \hat{\beta}^T \end{bmatrix}^T$ for which (3.3) is satisfied for each k. Therefore, since $\mathbf{D}_N$ captures inputs and outputs of the system (from its definition), we have that

$$\mathbf{D}_N \cdot \hat{\theta} = \mathbf{0} = \mathbf{D}_N \cdot \theta \Rightarrow \mathbf{D}_N \cdot (\theta - \hat{\theta}) = \mathbf{0}. \tag{3.17}$$

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 (3.17), 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 (3.16) holds so does (3.3), 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 (3.3). 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 (3.3) and its corresponding ACSL annotation (3.6) 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 (3.16) 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 $\mathbf{D}^y_{2n+1}$.

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 (3.16) as *instantiation-based invariants.*

3.2.1 Defining Instantiation-Based Invariants as Code Annotation

Similarly to the IO controller invariants from (3.3), 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 (3.16) can be captured as the code annotation from (3.18), 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 3.2.

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,

$$\begin{array}{l}\texttt{/*@ assert ((\textbackslash at(y,}k_0\texttt{)==}y_0\texttt{)\&\&...\&\&(\textbackslash at(y,}k_{n-1}\texttt{)==}y_{n-1}\texttt{) \&\&}\\ \quad\texttt{@ (\textbackslash at(u,}k_0\texttt{)==}u_0\texttt{)\&\&...\&\&(\textbackslash at(u,}k_{3n}\texttt{)==}u_{3n}\texttt{))}\\ \quad\texttt{@ }\Rightarrow\texttt{ ((\textbackslash at(y,}k_n\texttt{) == }y_n\texttt{) \&\& ... \&\& (\textbackslash at(y,}k_{3n}\texttt{)==}y_{3n}\texttt{)) */}\end{array} \tag{3.18}$$

$$\begin{array}{l}\texttt{/*@ assert ((\textbackslash at(y,}k_0\texttt{)==}y_0\texttt{)\&\&...\&\&(\textbackslash at(y,}k_{n-1}\texttt{)==}y_{n-1}\texttt{))} \Rightarrow\\ \quad\texttt{@ ((\textbackslash at(y,}k_n\texttt{)==}y_n\texttt{)\&\&...\&\&(\textbackslash at(y,}k_{3n}\texttt{)==}y_{3n}\texttt{)) */}\end{array} \tag{3.19}$$

whose form is shown in (3.19). In Section 3.3.1, we will compare efficiency of these approaches.

Remark. *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 [64]) 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 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* (3.18)*, or introduce an additional assignment* $\mathbf{z}_{n-1} = 0$ *in the* `verify_driver` *function with an* `assert` *statement similar to the one in* (3.19)*.*

```
extern double input();

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

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

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

Listing 4: One structure of the code annotations for verification of the IO only conformance using the *Instantiation-based Invariants* from (3.16); Note that u_t denotes u_t from the matrix $\mathbf{D}_{2n+1}$

3.2.2 Instantiation-Based Invariants for Inexact Controller Implementations

The invariant introduced in (3.16) 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 3.1.5 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 (3.10). Thus, from (3.16) we have that $\mathbf{D}_{2n+1} \cdot \hat{\theta} = \mathbf{0}$ which is (as in Proposition 1) equivalent to the invariant in (3.11).

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

$$\mathbf{D}_{2n+1} \cdot \theta + \mathbf{D}_{2n+1} \cdot \Delta\theta = \mathbf{0}. \tag{3.20}$$

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

$$\begin{aligned}\exists\Delta\beta_i, \Delta\alpha_i \in \mathbb{R}, i = 0, ..., n, |\Delta\beta_i| \leq \epsilon_\beta \wedge |\Delta\alpha_i| \leq \epsilon_\alpha \wedge \\ \mathbf{D}^y_{2n+1}\Delta\alpha + \mathbf{D}^u_{2n+1}\Delta\beta = \mathbf{v} \ \wedge \\ y_{n+i} = \mathbf{D}^y_{2n+1}(n+i), \quad i = 0, ..., 2n,\end{aligned} \tag{3.21}$$

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 (3.14) (for the exact controller specification).

```
/*@ assert \exists real $a_0, ..., a_{n-1}, b_0, ..., b_n$
  @ ($a_0 \le \epsilon_\alpha$)&&($a_0 \ge -\epsilon_\alpha$)&&...&&($a_{n-1} \le \epsilon_\alpha$)&&($a_{n-1} \ge -\epsilon_\alpha$)&&
  @ ($b_0 \le \epsilon_\beta$)&&($b_0 \ge -\epsilon_\beta$)&&...&&($b_n \le \epsilon_\beta$)&&($b_n \ge -\epsilon_\beta$) &&
  @ ((\at(y,$k_0$)==$y_0$) &&...&& (\at(y,$k_{n-1}$)==$y_{n-1}$) &&
  @ (\at(u,$k_0$) ==$u_0$) &&...&& (\at(u,$k_{3n}$)==$u_{3n}$))
  @ $\Rightarrow$ ((\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) ) */
```
(3.22)

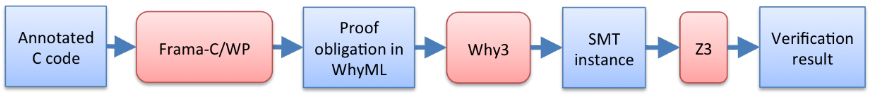

Figure 3.1: The verification toolchain of the invariant checking-based approach.

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 (3.18), for exact controller implementations, is introduced in (3.22). 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`.

3.3 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. 5.1).

To automatically verify C code annotated with ACSL specification [10], we employ the popular software verification platform Frama-C [22]. We also exploit WP [9], 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 [13], both automatic theorem provers (e.g., Z3 [26]) or interactive theorem provers (e.g., Coq [7]). 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. 5.1, 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 3.1 and 3.2 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). Moreover, 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.

3.3.1 Evaluation

To evaluate the developed verification framework, we first considered the controller specification (i.e., model) from (3.4). 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 (3.4). 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 (3.7), with transfer function (3.8). Finally, we exploited both types of `assert` statements from (3.18) and (3.19) to specify instantiation based invariants; we verified IO conformance of the example controller Σ from (3.4) and the inexact controller $\tilde{\Sigma}$ from (3.7) using the invariant (3.22).

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 devel-

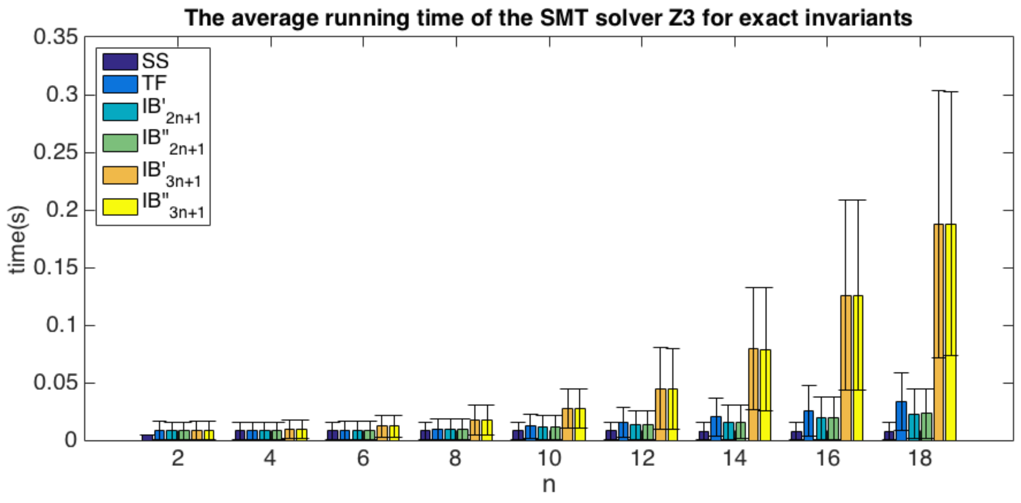

Figure 3.2: Z3 running times for LTI controller verification using five different types of controller invariants.

oped 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 (6.1); (b) IO only invariants based on the transfer function (denoted by TF), defined in (3.3) and (3.6), (c) Instantiation-based IO invariants defined in (3.16) and (3.18) (referred to as IB'_{3n+1}), (d) Instantiation-based IO invariants defined in (3.16) and (3.19) (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 3.2.1 on (3.18) – referred to as IB'_{2n+1}, (f) Instantiation-based IO in-

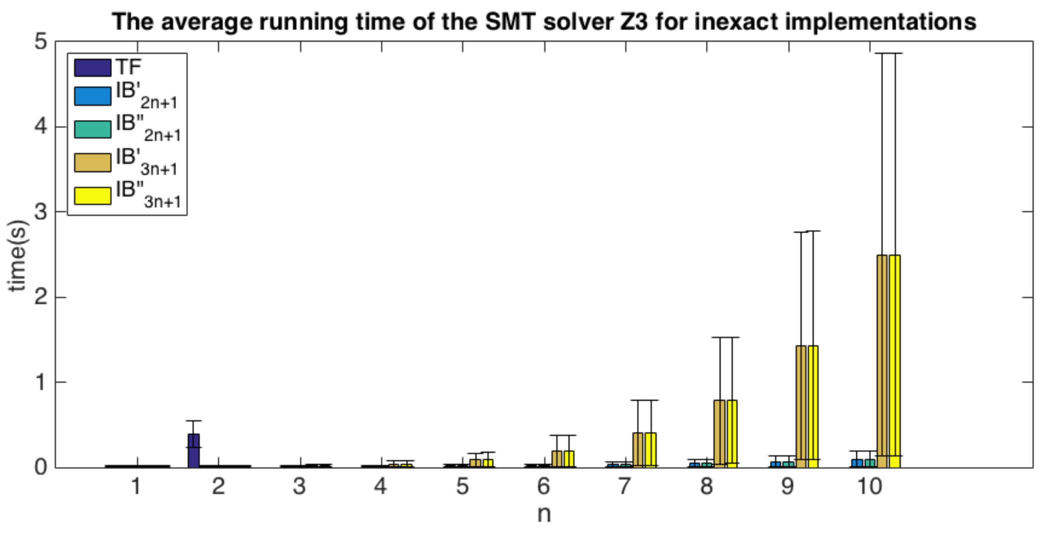

Figure 3.3: Z3 running times for verification of LTI controllers using 'inexact' invariants for all five different types of controller invariants. Note that in this case, verification of TF invariants does not scale well because controllers with the size greater than two can not be verified.

variants for only $2n+1$ points when the state variable is known and based, as described in Remark 3.2.1, on (3.19) – 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. 3.2 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. 3.2 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. 3.3. 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 3.1. 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}).

Finally, we analyzed the amount of time used by each tool in the verification framework. As shown in Tables 3.2 and 3.1 most of the verification time is used by Frama-C generating proof obligations in WhyML. Interestingly, running times for Z3 (shown in Fig. 3.2 and Fig. 3.3) present less than 5% of the overall verification times.

	SS	TF	IB'_{2n+1}	IB''_{2n+1}	IB'_{3n+1}	IB''_{3n+1}
Frama-C	76.6%	61.2%	51.8%	54.4%	49.8%	51.7%
Why3	20.5%	37.3%	47.7%	45.1%	49.6%	47.7%
Z3	2.8%	1.5%	0.5%	0.5%	0.6%	0.6%
Total time	0.3(s)	0.8(s)	2.1(s)	2.2(s)	4.6(s)	4.8(s)

Table 3.1: Percentage of time used by each tool in the verification framework for verification of controllers of size $n = 10$ with inexact implementations.

	SS	TF	IB'_{2n+1}	IB''_{2n+1}	IB'_{3n+1}	IB''_{3n+1}
Frama-C	77.6%	66.5%	55.2%	57.9%	56.0%	58.5%
Why3	20.2%	32.9%	44.7%	42.0%	43.7%	41.2%
Z3	2.2%	0.6%	0.1%	0.1%	0.3%	0.3%
Total time	0.4(s)	6.0(s)	27.4(s)	29.2(s)	56.7(s)	60.1(s)

Table 3.2: Percentage of time used by each tool in the verification framework for verification of controllers of size $n = 18$ with exact implementations.

Chapter 4

Similarity Checking-based Verification Approach

This chapter presents a similarity checking-based verification method [60]. This approach is based on extracting a model from the controller code and establishing equivalence between the original and the extracted models. Our technical approach relies on symbolic execution of the generated code. Symbolic expressions for state and output variables of the control function are used to reconstruct the model of the controller. The reconstructed model is then checked for input-output equivalence between the original and reconstructed model, using the well-known necessary and sufficient condition for the equivalence of two minimal LTI models. Verification is performed using real arithmetic. We account for some numerical errors by allowing for a bounded discrepancy between the models. We compare two approaches for checking the equivalence; one reduces the equivalence problem to an SMT problem, while the other uses a convex optimization formulation. We compare equivalence checking to an alternative verification approach introduced in Chapter 3, which converts the original LTI model into input-output based code annotations for verification at the code level.

This chapter is organized as follows: Section 4.1 presents model extraction from code, followed by the equivalence checking in Section 4.2. Section 4.3 evaluates the performance of the approaches.

4.1 Model Extraction from Linear Controller Implementation

In order to verify a linear controller implementation against its specification, we first extract an LTI model from the implementation (i.e., step function), and then compare it to the specification (i.e., the initial model). To obtain an LTI model from the step function, it is first necessary to identify the computation of the step function based on the program semantics. By the computation of a program, we mean how the execution of the program affects the global state.[1] This is also known as the *big-step transition relation* of a program, which is the relation between states before and after the execution of the program. In the next subsection, we explain how to identify the big-step transition relation of the step function via symbolic execution.

4.1.1 Symbolic Execution of Step Function

According to the symbolic execution semantics [47, 17, 14], we symbolically execute the step function with symbolic inputs and symbolic controller state.

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

When the execution is finished, we examine the effect of the step function on the global state where output and new controller state are produced as symbolic formulas.

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

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

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

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

For example, we consider symbolic execution for the step function in Listing 5 in Appendix A, obtained from the model (2.5), (2.6); we illustrate the transition equations of the step function as follows, replacing the original variable names with new shortened names for presentation purpose only, such as x for LTIS_DW.Internal_DSTATE, u for LTIS_U.u, and y for LTIS_Y.y:

$$\begin{aligned}
\mathtt{x[0]}^{(new)} &= ((0.87224 \cdot \mathtt{x[0]}) + ((0.822174 \cdot \mathtt{u[0]}) + (-0.438008 \cdot \mathtt{u[1]}))) \\
\mathtt{x[1]}^{(new)} &= ((0.366377 \cdot \mathtt{x[1]}) + ((-0.278536 \cdot \mathtt{u[0]}) + (-0.824312 \cdot \mathtt{u[1]}))) \\
\mathtt{x[2]}^{(new)} &= ((-0.540795 \cdot \mathtt{x[2]}) + ((0.874484 \cdot \mathtt{u[0]}) + (0.858857 \cdot \mathtt{u[1]}))) \\
\mathtt{x[3]}^{(new)} &= ((-0.332664 \cdot \mathtt{x[3]}) + ((-0.117628 \cdot \mathtt{u[0]}) + (-0.506362 \cdot \mathtt{u[1]}))) \\
\mathtt{x[4]}^{(new)} &= ((-0.204322 \cdot \mathtt{x[4]}) + ((-0.955459 \cdot \mathtt{u[0]}) + (-0.622498 \cdot \mathtt{u[1]}))) \\
\mathtt{y[0]}^{(new)} &= (((((-0.793176 \cdot \mathtt{x[0]}) + (0.154365 \cdot \mathtt{x[1]})) + (-0.377883 \cdot \mathtt{x[2]})) \\
&\quad +(-0.360608 \cdot \mathtt{x[3]})) + (-0.142123 \cdot \mathtt{x[4]})) \\
\mathtt{y[1]}^{(new)} &= (((((0.503767 \cdot \mathtt{x[0]}) + (-0.573538 * \cdot \mathtt{x[1]})) + (0.170245 \cdot \mathtt{x[2]})) \\
&\quad +(-0.583312 \cdot \mathtt{x[3]})) + (-0.56603 \cdot \mathtt{x[4]})).
\end{aligned} \tag{4.1}$$

4.1.2 Linear Time-Invariant System Model Extraction

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

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

For example, from the transition equations (4.1), `x[0]`, `x[1]`, `x[2]`, `x[3]` and `x[4]` are identified as controller state variables as given the input variables `u[0]` and `u[1]`, and the output variables `y[0]` and `y[1]`.

The next step is to convert the transition equations into a canonical form. We fully expand the expressions on the right-hand side of the transition equations using the distributive law. The resulting expressions are represented in the form of the sum of products without containing any parentheses. We check if the expressions equations are linear (i.e., each product term should be the multiplication of a constant and a single variable), and otherwise, it is rejected. Finally, each transition equation is represented as the following canonical form

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

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

Since the transition equations in canonical form are a system of linear equations, we finally rewrite the transition equations as matrix equations. In order to do this, we first define the input variable vector $\mathbf{u} = vec(V_{input})$, the output variable vector $\mathbf{y} = vec(V_{output})$ and the state variable vector $\mathbf{x} = vec(V_{state})$ where $vec(V)$ denotes the vectorization of the set V (e.g.,

$vec(\{v_1, v_2, v_3\}) = [v_1, v_2, v_3]^{\mathrm{T}}$). This allows for rewriting each transition equation in terms of the state variable vector $\mathbf{x}$ and the input variable vector $\mathbf{u}$ as

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

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

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

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

For example, consider the transition equation about $\texttt{y[0]}^{(new)}$ in (4.1), which is represented in canonical form, and then rewritten as a vector equation (i.e., equation in terms of the state and the input variable vectors) as follows

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

where $\mathbf{x} = [\texttt{x[0]}, \texttt{x[1]}, \texttt{x[2]}, \texttt{x[3]}, \texttt{x[4]}]^{\mathrm{T}}$, and $\mathbf{u} = [\texttt{u[0]}, \texttt{u[1]}]^{\mathrm{T}}$. Converting each transition equation (4.1) into the corresponding vector equation, we finally reconstruct the LTI model (i.e., same as (2.5) (2.6)) from the step function of Listing 5 in Appendix A.

Remark. *In general, the size of the extracted model* $\hat{\mathbf{\Sigma}}$ *may not be equal to the size of the initial controller model* $\mathbf{\Sigma}$ *from* (6.1) *(i.e., n). As we assume that* $\mathbf{\Sigma}$ *is minimal, if the obtained model has the size less than n it would clearly have to violate input-output (IO) requirements of the controller. However, if the size of* $\hat{\mathbf{\Sigma}}$ *is larger than n, we consider a controllable and observable subsystem computed via Kalman decomposition [64] from the extracted model, as the* $\hat{\mathbf{\Sigma}}(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \hat{\mathbf{D}})$ *model extracted from the code. Note that* $\hat{\mathbf{\Sigma}}$ *is minimal in this case, and thus its size has to be equal to n to provide IO conformance with the initial model.*

4.2 Input-Output Equivalence Checking between Linear Controller Models

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

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

such that

$$\hat{\mathbf{A}} = \mathbf{T}\mathbf{A}\mathbf{T}^{-1}, \qquad \hat{\mathbf{B}} = \mathbf{T}\mathbf{B}, \qquad \hat{\mathbf{C}} = \mathbf{C}\mathbf{T}^{-1}, \qquad \text{and} \qquad \hat{\mathbf{D}} = \mathbf{D} \tag{4.2}$$

where $\mathbf{T}$ is referred to as the *similarity transformation matrix* [64]. Thus, given two minimal LTI models, the problem of equivalence checking between the models is reduced to the problem of finding a similarity transformation matrix for the models. The rest of this section explains how to formulate this problem as a satisfiability problem and a convex optimization problem.

4.2.1 Satisfiability Problem Formulation

We start by describing an approach to formulate the problem of finding similarity transformation matrices as the satisfiability problem instance when two LTI models $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ and $\hat{\mathbf{\Sigma}}(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \hat{\mathbf{D}})$ are given. Since existing SMT solvers hardly support matrices and linear algebra operations, we encode the similarity transformation matrix $\mathbf{T}$ as a set of scalar variables $\{T_{i,j} \mid 1 \leq i, j \leq n\}$ where $T_{i,j}$ is the variable to represent the element in the i-th row and j-th column of the matrix $\mathbf{T}$. The following constraints rephrase the equations of (4.2) in an element-wise manner

$$\bigwedge_{1\leq i\leq n} \bigwedge_{1\leq j\leq n} \left(\sum_{1\leq k\leq n} \hat{A}_{i,k} T_{k,j} = \sum_{1\leq k\leq n} T_{i,k} A_{k,j} \right) \wedge \bigwedge_{1\leq i\leq n} \bigwedge_{1\leq j\leq n} \left(\hat{B}_{i,j} = \sum_{1\leq k\leq n} T_{i,k} B_{k,j} \right)$$
$$\bigwedge_{1\leq i\leq n} \bigwedge_{1\leq j\leq n} \left(\sum_{1\leq k\leq n} \hat{C}_{i,k} T_{k,j} = C_{i,j} \right) \wedge \bigwedge_{1\leq i\leq n} \bigwedge_{1\leq j\leq n} \hat{D}_{i,j} = D_{i,j} \tag{4.3}$$

It is important to highlight that although a similarity transform always results in an IO equivalent new controller, due to finite-precision computation of the code generator performing controller optimization, it is expected that the produced controller will slightly differ from a controller that is similar to the initial controller. Consequently, there is a need to extend our input-output invariants for the case with imprecise specification of the similarity transform. To achieve this, given error bound ϵ, the following constraints extends (4.3) to tolerate errors up to error bound ϵ

$$\begin{gathered}
\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq n} -\epsilon \leq \left(\sum_{1\leq k\leq n} \hat{A}_{i,k}T_{k,j}\right) - \left(\sum_{1\leq k\leq n} T_{i,k}A_{k,j}\right) \leq \epsilon \\
\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq n} -\epsilon \leq \hat{B}_{i,j} - \left(\sum_{1\leq k\leq n} T_{i,k}B_{k,j}\right) \leq \epsilon \\
\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq n} -\epsilon \leq \left(\sum_{1\leq k\leq n} \hat{C}_{i,k}T_{k,j}\right) - C_{i,j} \leq \epsilon \\
\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq n} -\epsilon \leq \hat{D}_{i,j} - D_{i,j} \leq \epsilon
\end{gathered} \tag{4.4}$$

For example, suppose that the original LTI model $\mathbf{\Sigma}(\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D})$ from (2.3)(2.4), the reconstructed model from the implementation $\hat{\mathbf{\Sigma}}(\hat{\mathbf{A}},\hat{\mathbf{B}},\hat{\mathbf{C}},\hat{\mathbf{D}})$ from (2.5)(2.6) and the error bound $\epsilon = 10^{-6}$ are given. Having the problem instance formulated as (4.4), the similarity transformation matrix $\mathbf{T}$ for those models can be found using an SMT solver which supports the quantifier-free linear real arithmetic, QF_LRA for short. Due to the lack of space, only the first row of $\mathbf{T}$ is shown here

$$
\begin{aligned}
T_{1,1} &= -\frac{445681907965836469807842159338}{818667375305282643804030465563} \quad (\approx -0.544399156750667)\\
T_{1,2} &= -\frac{135442022883031921128620509482}{818667375305282643804030465563} \quad (\approx -0.165442059801384)\\
T_{1,3} &= \frac{198172776374831449251211655628}{818667375305282643804030465563} \quad (\approx 0.242067461044165)\\
T_{1,4} &= -\frac{351256050550998919211978953100}{818667375305282643804030465563} \quad (\approx -0.429058064513855)\\
T_{1,5} &= -\frac{476345345040634696989970420590}{818667375305282643804030465563} \quad (\approx -0.581854284748456)
\end{aligned}
$$

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

4.2.2 Convex Optimization Problem Formulation

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

$$
\begin{aligned}
&\text{variables} && e \in \mathbb{R}, \mathbf{T} \in \mathbb{R}^{n\times n} \\
&\text{minimize} && e \\
&\text{subject to} && \epsilon \le e, \\
& && \left\|\hat{\mathbf{A}}\mathbf{T} - \mathbf{T}\mathbf{A}\right\|_{\infty} \le e,\ \left\|\hat{\mathbf{B}} - \mathbf{T}\mathbf{B}\right\|_{\infty} \le e, \\
& && \left\|\hat{\mathbf{C}}\mathbf{T} - \mathbf{C}\right\|_{\infty} \le e,\ \left\|\hat{\mathbf{D}} - \mathbf{D}\right\|_{\infty} \le e
\end{aligned}
\tag{4.5}
$$

For example, given two LTI models $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ from (2.3)(2.4) and $\hat{\mathbf{\Sigma}}(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \hat{\mathbf{D}})$ from (2.5)(2.6) and the error bound $\epsilon = 10^{-6}$, by (5.16), the similarity transformation matrix $\mathbf{T}$ can be found using the convex optimization solver CVX as follows

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

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

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

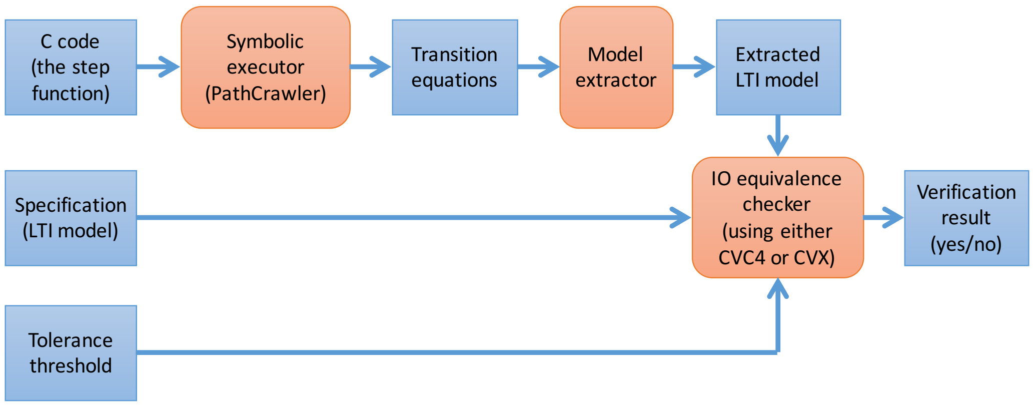
C code (the step function)
Symbolic executor (PathCrawler)
Transition equations
Model extractor
Extracted LTI model
Specification (LTI model)
IO equivalence checker (using either CVC4 or CVX)
Verification result (yes/no)
Tolerance threshold

Figure 4.1: The verification toolchain for the similarity checking-based approach.

resulting in the difference between two matrices equal to

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

4.3 Evaluation

To evaluate our verification approach described in Section 4.1 and Section 4.2, we compared it to our earlier work based on invariant checking in Chapter 3.

4.3.1 Verification Toolchain

We implemented an automatic verification framework (presented in Fig. 5.1) based on the proposed approach described in Section 4.1 and Section 4.2.

We refer to this approach as similarity checking (SC)-based approach. Given a step function (i.e., C code), we employ the off-the-shelf symbolic execution tool PathCrawler [83] to symbolically execute the step function and generate a set of transition equations. The model extractor which implements the method in Section 4.1.2 extracts an LTI model from the transition equations. Finally, the equivalence checker based on the method in Section 4.2 decides the similarity between the extracted LTI model and the given specification (i.e., LTI model), and produces the verification result. The equivalence checker uses either the SMT solver CVC4 [8][2] or the convex optimization solver CVX [40] depending on the formulation employed, which is described in Section 4.2.

For the invariant checking (IC)-based approach described in Chapter 3, we use the toolchain Frama-C/Why3/Z3 to verify C code with annotated controller invariants, as described in Chapter 3. The step function is annotated with the invariants as described in Chapter 3. Given annotated C code, Frama-C/Why3 [22, 13] generates proof obligations as SMT instances. The SMT solver Z3 [26][3] solves the proof obligations and produces the verification result (see Chapter 3 for more details).

4.3.2 Scalability Evaluation

To evaluate the SC-based approach compared to the IC-based approach, we randomly generate stable linear controller specifications (i.e., the elements

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

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

of $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$). Since we observed that the controller dimension n dominates the performance (i.e., running time) of both approaches, we vary n from 2 to 14, and generate three controller specifications for each n. For each controller specification, we employ the code generator Embedded Coder to generate the step function in C. Since we use the LTI system block of Simulink for code generation, the structure of generated C code is not straightforward, having multiple loops and pointer arithmetic operations as illustrated in the step function [59]. This negatively affects the performance of the IC-based approach for reasons to be described later in this subsection. For a comparative evaluation, we use both SC-based and IC-based approaches to verify the generated step function C code against its specification. For each generated controller, we checked that IC-based and SC-based approaches give the same verification result, as long as both complete normally.

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

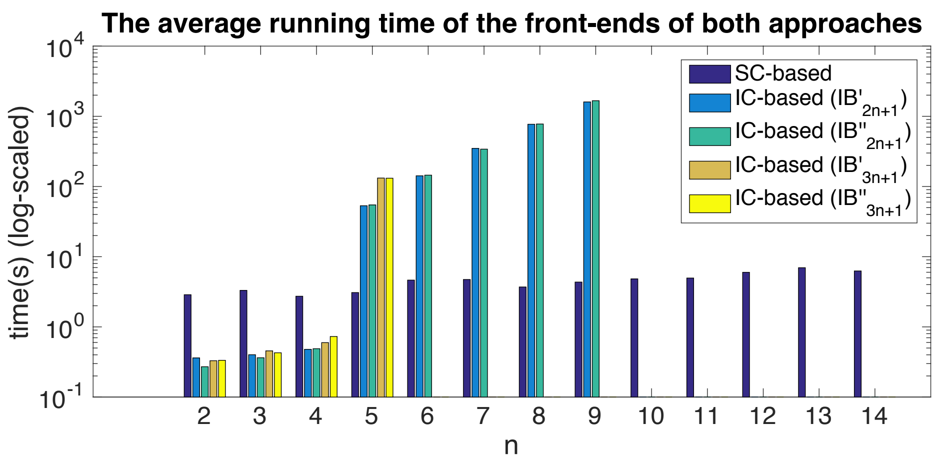

Figure 4.2: The average running time of the front-ends of both SC-based and IC-based approaches (with the log-scaled y-axis)

We first evaluate the frond-end of both approaches (i.e., the whole verification process until constraint solving). Fig. 4.2 shows that the average running time of the front-ends of both approaches, where missing bars indicate no data due to the lack of scalability of the utilized verification approach (e.g., the tool's abnormal termination or no termination for a prolonged time). Here, IB'_{2n+1}, IB''_{3n+1}, IB''_{3n+1} and IB'_{2n+1} denote the variations of annotating methods as described in Chapter 3. We observe that the running time of the IC-based approaches exponentially increase as the controller dimension n increases, while the SC-based approach remains scalable. The main reason for this is that the IC-based approach requires the preprocessing of code (as described in Chapter 3), which is unrolling the execution of the step function multiple times (e.g., $2n + 1$ or $3n + 1$ times) as well as unrolling each loop in the step function $(n + 1)$ times. Therefore, in contrast with the SC-based approach, the IC-based approach needs to handle the significantly increased lines of code due to unrolling, so it does not scale up.

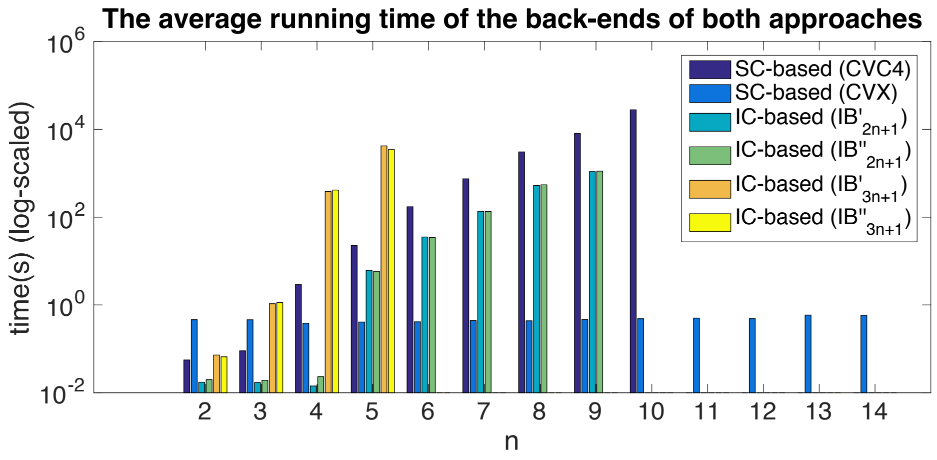

Figure 4.3: The average running time of the back-ends of both SC-based and IC-based approaches (with the log-scaled y-axis)

Next, we evaluate the back-end of both approaches (i.e., constraint solving). Fig. 4.3 shows the average running time of the back-ends of both approaches, where missing bars result from the lack of scalability of either the constraint solver used at this stage or the front-end tools. "SC-based (CVC4)" denotes the SMT-based formulation while "SC-based (CVX)" denotes the convex optimization-based formulation. Recall that the SC-based approach using CVC4 and the IC-based approaches employ the SMT solvers for constraint solving, which uses the arbitrary-precision arithmetic. We observe that the running time of the back-ends of those approaches exponentially increase as the controller dimension n increases because of the cost of the bignum arithmetic, while the SC-based approach using CVX remains scalable.

Chapter 5

Verification of Finite-Precision Controller Software

This chapter describes an extended method [61] that builds on the similarity checking-based method in Chapter 4 in order to verify finite-precision controller software considering the effect of floating-point arithmetic. In Chapter 3 and Chapter 4, 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 chapter, we build on the work of Chapter 4, 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 the last chapter (Chapter 4), 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 chapter is organized as follows: Section 5.1 describes how to extract a model from the controller code. Section 5.2 presents the approximate equivalence checking. Section 5.3 evaluates the scalability of our approach.

5.1 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 the method in Chapter 4 to consider the quantization error in the finite-precision implementation. To obtain a model from the step function, we employ the symbolic execution technique [17, 47], 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 (6.1). We will describe the representation of extracted models in the next subsection.

5.1.1 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 (6.1) with the new terms of absolute errors and relative errors, and obtain the following representation of quantized controller model:

$$\begin{aligned}\hat{\mathbf{z}}_{k+1} &= (\hat{\mathbf{A}} + \mathbf{E_A})\hat{\mathbf{z}}_k + (\hat{\mathbf{B}} + \mathbf{E_B})\mathbf{u}_k + \mathbf{e_z} \\ \mathbf{y}_k &= (\hat{\mathbf{C}} + \mathbf{E_C})\hat{\mathbf{z}}_k + (\hat{\mathbf{D}} + \mathbf{E_D})\mathbf{u}_k + \mathbf{e_y}.\end{aligned} \tag{5.1}$$

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.

5.1.2 Symbolic Execution of Floating-Point Controller Implementation

In our approach, the symbolic execution technique [17, 47] 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 (as described in Chapter 4).

However, unlike the approach in Chapter 4, this 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 [14]) 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*:

$$v^{(new)} = f(v_1, v_2, \dots, v_t) \tag{5.2}$$

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. $f(v_1, v_2, \dots, v_t)$ 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 [59] , which is generated by Embedded Coder (the code generator of MATLAB/Simulink) for the LTI controller models (2.5)(2.6). We illustrate one of the transition equations obtained from the symbolic execution of the step function as follows:

$$\begin{aligned} \mathtt{y[1]}^{(new)} &= (((((0.503767 \otimes \mathtt{x[0]}) \oplus (-0.573538 \otimes \mathtt{x[1]})) \oplus (0.170245 \otimes \mathtt{x[2]})) \\ &\quad \oplus(-0.583312 \otimes \mathtt{x[3]})) \oplus (-0.56603 \otimes \mathtt{x[4]})). \end{aligned} \tag{5.3}$$

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 (5.11) from the symbolic expressions.

5.1.3 Quantization Error Analysis and Model Extraction

This subsection explains how to extract the quantized controller model (5.11) from a set of symbolic expressions (5.2) 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 the following syntax, thus guaranteeing the linearity:

$$
\begin{aligned}
&f := v \mid f \oplus f \mid f \ominus f \mid f \circledast f_c \mid f_c \circledast f \\
&f_c := c \mid f_c \circledast f_c
\end{aligned}
$$

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 [35, 60] 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 equation (5.2) can be represented in the following canonical form presented earlier in Chapter 4:

$$v^{(new)} = \sum_{i=1}^{t} c_i v_i \tag{5.4}$$

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 [2] views a finite precision floating-point operation as the corresponding real operation followed by a rounding operation:

$$x_1 \circledast x_2 = rnd(x_1 * x_2) \tag{5.5}$$

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 [2]. A rounding operator can be modeled as follows [38]:

$$rnd(x) = x(1 + e) + d \tag{5.6}$$

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 [68]. Combining the two equations (5.5) and (5.6), we have the following model for the floating-point operations:

$$x_1 \circledast x_2 = (x_1 * x_2)(1 + e) + d \tag{5.7}$$

After rewriting the symbolic expression of the transition equation (5.2) applying the equation (5.7), suppose that we have the following equation form:

$$v^{(new)} = \sum c_i v_i + err_{rel} + err_{abs} \tag{5.8}$$

where $\sum c_i v_i$ is the exact expression as (5.4), 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:

$$\begin{aligned} v^{(new)} =& \sum c_i v_i + \sum err_i v_i + err_{abs} \\ =& \sum c_i v_i + err \sum v_i + err_{abs} \end{aligned} \tag{5.9}$$

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:

$$\begin{aligned} v^{(new)} =& [c_1, c_2, ..., c_n]\mathbf{x} + [err, err, ..., err]\mathbf{x} \\ &+ [c'_1, c'_2, ..., c'_p]\mathbf{u} + [err, err, ..., err]\mathbf{u} + err_{abs} \end{aligned} \tag{5.10}$$

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:

$$\begin{aligned}\mathbf{x}^{(new)} &= (\hat{\mathbf{A}} + \mathbf{E}_\mathbf{A})\mathbf{x} + (\hat{\mathbf{B}} + \mathbf{E}_\mathbf{B})\mathbf{u} + \mathbf{e}_\mathbf{x} \\ \mathbf{y}^{(new)} &= (\hat{\mathbf{C}} + \mathbf{E}_\mathbf{C})\mathbf{x} + (\hat{\mathbf{D}} + \mathbf{E}_\mathbf{D})\mathbf{u} + \mathbf{e}_\mathbf{y}.\end{aligned} \tag{5.11}$$

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 $\|\mathbf{E}_\mathbf{A}\|, \|\mathbf{E}_\mathbf{B}\|, \|\mathbf{E}_\mathbf{C}\|, \|\mathbf{E}_\mathbf{D}\| \leq b^*_{rel}$. The absolute error vectors $\mathbf{e}_\mathbf{x}$ and $\mathbf{e}_\mathbf{y}$ are bounded by b^*_{abs} such that $\|\mathbf{e}_\mathbf{x}\|, \|\mathbf{e}_\mathbf{y}\| \leq b^*_{abs}$. 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 (5.3), from which via the floating-point error analysis, we have:

$$\begin{aligned}\texttt{y[1]}^{(new)} &= (((((0.503767 \otimes \texttt{x[0]}) \oplus (-0.573538 \otimes \texttt{x[1]})) \oplus (0.170245 \otimes \texttt{x[2]})) \\ &\quad \oplus(-0.583312 \otimes \texttt{x[3]})) \oplus (-0.56603 \otimes \texttt{x[4]})) \\ &= 0.503767 \cdot \texttt{x[0]} + -0.573538 \cdot \texttt{x[1]} + 0.170245 \cdot \texttt{x[2]} \\ &\quad + -0.583312 \cdot \texttt{x[3]} + -0.56603 \cdot \texttt{x[4]} + err_{rel} + err_{abs} \\ &= 0.503767 \cdot \texttt{x[0]} + -0.573538 \cdot \texttt{x[1]} + 0.170245 \cdot \texttt{x[2]} \\ &\quad + -0.583312 \cdot \texttt{x[3]} + -0.56603 \cdot \texttt{x[4]} \\ &\quad + err(\texttt{x[0]} + \texttt{x[1]} + \texttt{x[2]} + \texttt{x[3]} + \texttt{x[4]}) + err_{abs}\end{aligned} \tag{5.12}$$

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 $b_{abs} \approx 1.235164 \times 10^{-323}$.

5.2 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 (5.11) and the initial model (6.1) with a notion of approximate input-output (IO) equivalence.

5.2.1 Approximate Input-Output Equivalence

This subsection defines an approximate IO equivalence relation, inspired by the similarity transformation of LTI systems [64]. 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* [64]. Assuming that a proper $\mathbf{T}$ is given, we substitute $\mathbf{z}_k$ by $\mathbf{T}\hat{\mathbf{z}}$ in the initial LTI model (6.1), thus having:

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

or

$$\hat{\mathbf{z}}_{k+1} = (\mathbf{T}^{-1}\mathbf{A}\mathbf{T})\hat{\mathbf{z}}_k + (\mathbf{T}^{-1}\mathbf{B})\mathbf{u}_k, \quad \mathbf{y}_k = (\mathbf{C}\mathbf{T})\hat{\mathbf{z}}_k + \mathbf{D}\mathbf{u}_k. \tag{5.13}$$

By the similarity transformation, two LTI systems (6.1) and (5.13) are *similar*, meaning that they are IO equivalent. We now compare the transformed initial LTI model (5.13) and the quantized controller model (5.11) that is extracted from the step function. Equating the corresponding coefficient matrices of the two models (5.13) and (5.11), we have:

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

or

$$\mathbf{A}\mathbf{T} = \mathbf{T}\hat{\mathbf{A}} + \mathbf{T}\mathbf{E_A}, \quad \mathbf{B} = \mathbf{T}\hat{\mathbf{B}} + \mathbf{T}\mathbf{E_B}, \quad \mathbf{C}\mathbf{T} = \hat{\mathbf{C}} + \mathbf{E_C}, \quad \mathbf{D} = \hat{\mathbf{D}} + \mathbf{E_D} \tag{5.14}$$

However, the equality of the exact equivalence condition (5.14) 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 equations (5.14) are relaxed as follows:

$$\mathbf{A}\mathbf{T} \approx_\rho \mathbf{T}\hat{\mathbf{A}} + \mathbf{T}\mathbf{E_A}, \quad \mathbf{B} \approx_\rho \mathbf{T}\hat{\mathbf{B}} + \mathbf{T}\mathbf{E_B}, \quad \mathbf{C}\mathbf{T} \approx_\rho \hat{\mathbf{C}} + \mathbf{E_C}, \quad \mathbf{D} \approx_\rho \hat{\mathbf{D}} + \mathbf{E_D} \tag{5.15}$$

Finally, we say that the initial LTI model (6.1) and the quantized model (5.11) extracted from the implementation are approximately IO equivalent with precision ρ if there exists a similarity transformation matrix $\mathbf{T}$ which satisfies (5.15), 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.

5.2.2 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:

$$\exists \mathbf{T} : \forall \mathbf{E_A}, \mathbf{E_B}, \mathbf{E_C}, \mathbf{E_D} : \left\|\mathbf{E_A}\right\|, \left\|\mathbf{E_B}\right\|, \left\|\mathbf{E_C}\right\|, \left\|\mathbf{E_D}\right\| \leq b_{rel} \implies \text{(5.15) holds}$$

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 (5.15). 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.

5.2.3 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 (5.15) inappropriate to be formulated as a convex optimization problem, our approach is to derive a sufficient condition for (5.15). By over-approximating the error terms and removing the error variables, we derive such a sufficient condition for (5.15) which is formulated as a convex optimization problem as follows:

$$\begin{aligned} &\text{variables} && e \in \mathbb{R}, \mathbf{T} \in \mathbb{R}^{n\times n} \\ &\text{minimize} && e \\ &\text{subject to} && \left\|\hat{\mathbf{A}}\mathbf{T} - \mathbf{T}\mathbf{A}\right\|_\infty + n^2 \left\|\mathbf{T}\right\|_\infty b_{rel} \le e \\ &&& \left\|\hat{\mathbf{B}} - \mathbf{T}\mathbf{B}\right\|_\infty + n^2 \left\|\mathbf{T}\right\|_\infty b_{rel} \le e \\ &&& \left\|\hat{\mathbf{C}}\mathbf{T} - \mathbf{C}\right\|_\infty + n \cdot b_{rel} \le e, \quad \left\|\hat{\mathbf{D}} - \mathbf{D}\right\|_\infty + n \cdot b_{rel} \le e \end{aligned} \tag{5.16}$$

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

Remark. *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 equiv-*

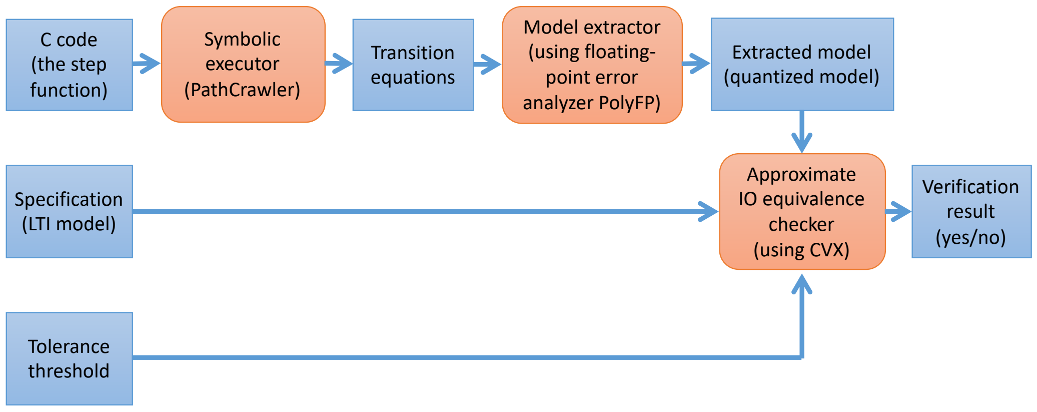
C code (the step function)
Symbolic executor (PathCrawler)
Transition equations
Model extractor (using floating-point error analyzer PolyFP)
Extracted model (quantized model)
Specification (LTI model)
Approximate IO equivalence checker (using CVX)
Verification result (yes/no)
Tolerance threshold

Figure 5.1: The verification toolchain

alence 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.

5.3 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 in Chapter 4 which assumes that the computations of controller implementations have no rounding errors.

5.3.1 Toolchain

This subsection presents the verification toolchain (shown in Fig. 5.1) that we implemented based on our method described in this chapter. The toolchain is an extension of the tool presented in Chapter 4 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 [83] to symbolically execute the step function and produce the transition equations for the step function. From the transition equations, the model extractor based on Section 5.1.3 extracts the quantized controller model using the floating-point error analysis tool PolyFP [3]. Finally, the extracted quantized model is compared with the given specification (i.e., LTI model) based on the approximate IO relation defined in Section 5.2. The approximate IO equivalence checker uses the convex optimization solver CVX [40] to solve the formulas in Section 5.2.3.

5.3.2 Scalability Analysis

This subsection evaluates the scalability of our approach/toolchain presented in this chapter. 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 [59]. 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

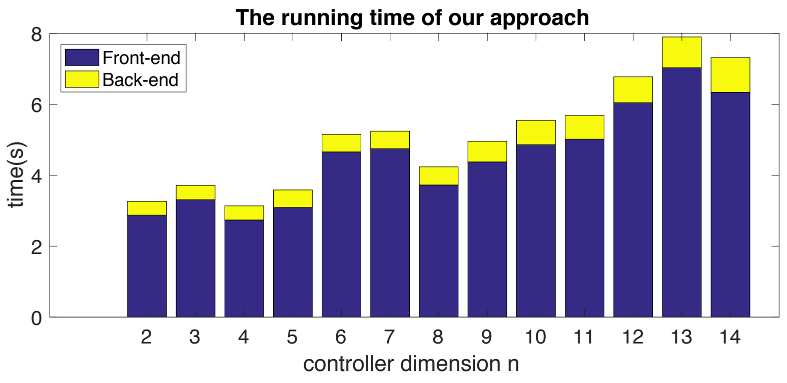

Figure 5.2: The running time of both the front-end and the back-end of our approach

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. 5.2, 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 described in Chapter 4 where the verification problem is simpler than our verification problem herein because the previous work in Chapter 4 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

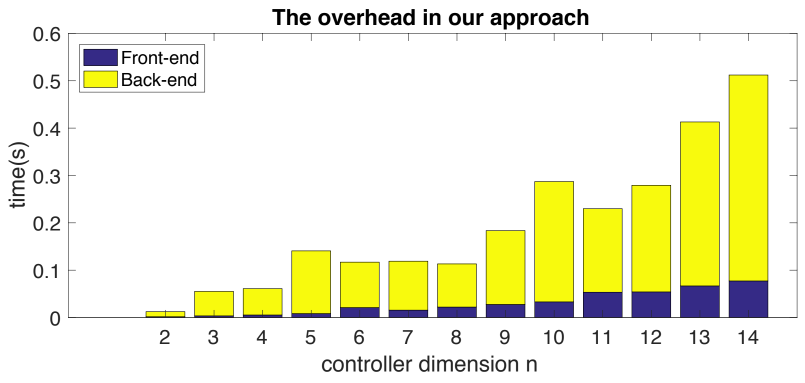

Figure 5.3: The overhead in both the front-end and the back-end of our approach

finite precision controller implementations considering the rounding errors in computation. Fig. 5.3 shows the computational 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 in Chapter 4 at a cost of providing higher assurance for the correctness of the finite precision computations of controller implementations.

Chapter 6

Linear Controller Verifier

This chapter describes LCV (Linear Controller Verifier), the prototype tool that implements our verification approaches in this dissertation. This chapter also evaluates the tool LCV through the case study and the scalability analysis.

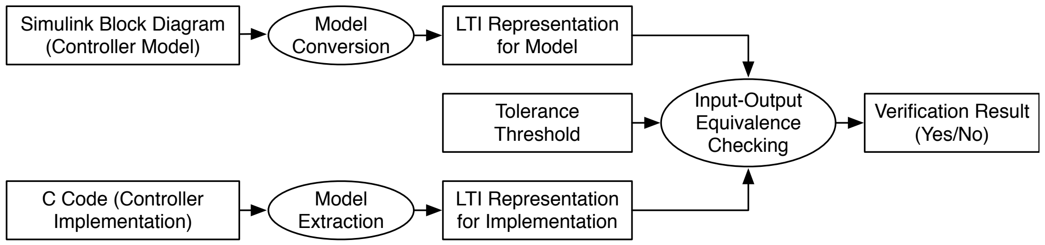
Simulink Block Diagram (Controller Model)
Model Conversion
LTI Representation for Model
Tolerance Threshold
Input-Output Equivalence Checking
Verification Result (Yes/No)
C Code (Controller Implementation)
Model Extraction
LTI Representation for Implementation

Figure 6.1: The verification flow of LCV.

6.1 Verification Flow of Linear Controller Verifier

This section describes the verification flow (shown in Fig. 6.1) and the implementation details of LCV. LCV takes as input a Simulink block diagram (i.e., controller model) and a C code (i.e., controller implementation). LCV assumes that the name of the step function (i.e., controller's entry function) is also given, and the step function interfaces through given global variables. In other words, the input(output) variables are declared in the global scope, and are written(read) before(after) the execution of the step function.[1]

LCV can handle any C program that has a deterministic and finite execution path for a symbolic input, which is often found to be true for many embedded linear controllers. Moreover, we observe that a Simulink block diagram of a LTI controller can be converted into a state-space representation form, thus being amenable to our verification methods. Thus, LCV can handle any Simulink block diagram which results in an LTI system (i.e., satisfying the superposition property). The block diagram may include basic blocks (e.g., constant block, gain block, sum block), subsystem blocks (i.e., hierarchy) and series/parallel/feedback connections of those blocks. Extending LCV to verify a broader class of controllers is an avenue for future work.

As the first step of the verification, the Simulink block diagram is converted into a state space representation of an LTI system, which is defined

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

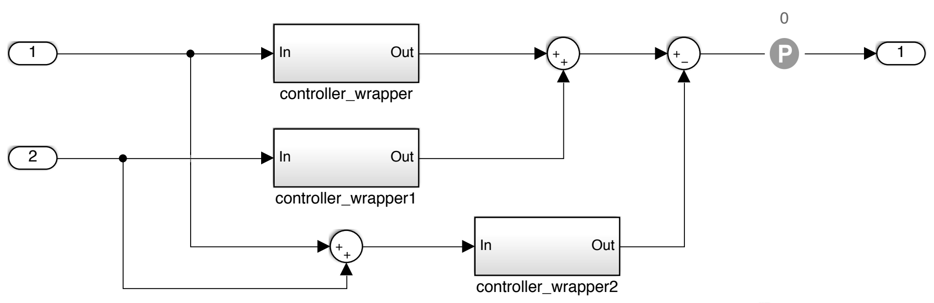

Figure 6.2: The simulink block diagram for checking the additivity of the controller

as follows:

$$\begin{aligned} \mathbf{z}_{k+1} &= \mathbf{A}\mathbf{z}_k + \mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{z}_k + \mathbf{D}\mathbf{u}_k. \end{aligned} \tag{6.1}$$

where $\mathbf{u}_k$, $\mathbf{y}_k$ and $\mathbf{z}_k$ are the input vector, the output vector and the state vector at time k respectively. The matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ are controller parameters. Matlab function `linearize` is used to obtain the LTI model from the Simulink block diagram. This step assumes that the block diagram represents a linear controller model. A systematic procedure can remove this assumption: one can check whether a given Simulink block diagram is linear (i.e., both additive and homogeneous) using Simulink Design Verifier [75], a model checker for Simulink. For example, to check if a controller block in Simulink is additive or not, as shown in Figure 6.2, one can create two additional duplicates of the controller block, generate two different input sequences, and check if the output of the controller in response to the sum of two inputs is equal to the sum of two outputs of the controllers in response the two inputs respectively. In Figure 6.2, `controller_wrapper` wraps the actual controller under test, and internally performs multiplexing and de-multiplexing to handle the multiple inputs and outputs of the controller. Simulink Design Verifier serves checking if this holds for all possible input sequences. However, a limitation of the current version of Simulink Design Verifier is that it does not support all Simulink blocks and does not properly handle non-linear cases. In these cases, alternatively, one can validate the linearity of controllers using simulation-based tests instead of model checking, which can be systematically done by Simulink Test [76]. This method is not limited by any types of Simulink blocks, and can effectively disprove the linearity of controllers for non-linear cases. However, this alternative method may not be as rigorous as the model-checking based method because the simulation-based test does not consider all possible input cases.

The next step is extracting the LTI model from the controller implementation C code. To do this, LCV uses the symbolic execution technique which allows us to identify the computation of the step function (i.e., C function which implements the controller). By the computation, we mean the big-step transition relation on global states between before and after the execution of the step function. The big-step transition relation is represented as symbolic formulas that describe how global variables for the controller's state and output are updated in terms of the old values of the global variables as the effect of the step function execution. From the transition relation, an LTI model for the controller implementation is extracted as explained in Chapter 4 and Chapter 5. LCV employs the off-the-shelf symbolic execution tool PathCrawler [83], which outputs in an XML format the symbolic execution paths and the path conditions of a given C program. The idea behind this symbolic execution step is that linear controller codes used for embedded systems generally have simple control flows for the sake of deterministic real-time behaviors (e.g., fixed upper bound of loops).

Finally, LCV performs the input-output equivalence checking between the LTI model obtained from the block diagram and the LTI model extracted from the C code implementation. We assume that a proper tolerance threshold is given by a control engineer as a result of robustness analysis. To do the input-output equivalence checking, we employ the notion of similarity transformation [64], which implies that two minimal LTI models are input-output equivalent if and only if they are *similar* to each other (i.e., there exists a similarity transformation matrix T that satisfies certain conditions). Thus, we first minimize both the extracted model and the original model via Kalman Decomposition [64] (Matlab function `minreal`). The input-output equivalence checking problem is reduced to the problem of finding the existence of T (i.e., similarity checking problem). LCV formulates the similarity checking problem as a convex optimization problem [60], and employs CVX [40] to find T. In the formulation, the equality relation is relaxed to tolerate the numerical errors that come from multiple sources (e.g., the controller parameters, the computation of the implementation, the verification process) so that any two quantities which are closer than a given ϵ are considered to be equal. ϵ is chosen to be 10^{-5} for the case study that we performed in the next section.

The output of LCV is as follows: First of all, when LCV fails to extract an LTI model from code, it tells the reason (e.g., non-deterministic execution paths for a symbolic input due to branching over a symbolic expression condition, non-linear arithmetic computation due to the use of trigonometric functions). Moreover, for the case of non-equivalent model and code, LCV provides the LTI models obtained from the Simulink block diagram model and the C code respectively, so that the user can simulate both of the models and easily find an input sequence that leads to a discrepancy between their output behaviors. Finally, for the case of equivalent model and code, LCV additionally provides a similarity transformation matrix between the two LTI models, which is the key evidence to prove the input-output equivalence between the model and code.

6.2 Evaluation

We evaluate LCV through conducting a case study using a standard PID controller and a controller used in a quadrotor. We also evaluate the scalability of LCV in the subsequent subsection. The instruction and software of LCV are available online for evaluation[2], which contains all of the files used for the evaluation of the tool in this section (i.e., Simulink block diagram and C code implementation).

6.2.1 Case Study

PID Controller

In our case study, we first consider a proportional-integral-derivative (PID) controller, which is a closed-loop feedback controller commonly used in various control systems (e.g., industrial control systems, robotics, automotive). A PID controller attempts to minimize the error value e_t over time which is defined as the difference between a reference point r_t (i.e., desired value) and a measurement value y_t (i.e., $e_t = r_t - y_t$). To do this, the PID controller

[2]`http://cis.upenn.edu/~park11/lcv.html`

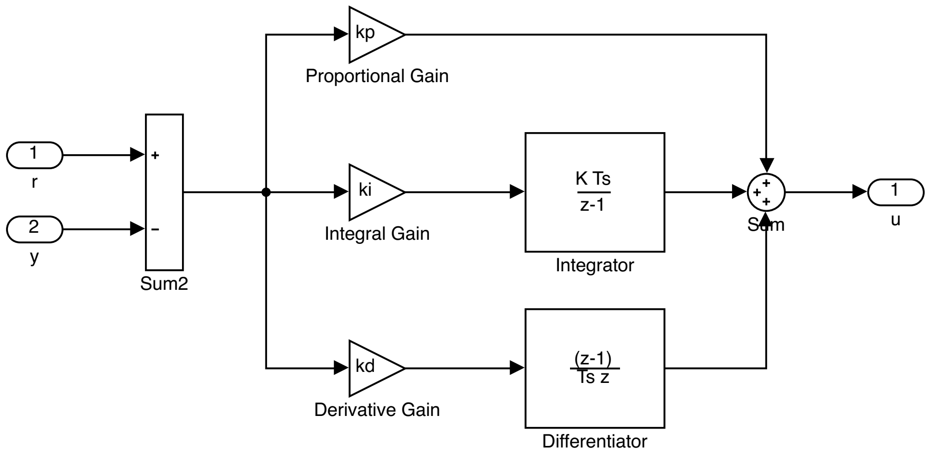

Figure 6.3: The block diagram of the PID controller.

adjusts a control input u_t computing the sum of the proportion term $k_p e_t$, integral term $k_i T \sum_{i=1}^{t} e_t$ and derivative term $k_d \frac{e_t - e_{t-1}}{T}$ so that

$$u_t = k_p e_t + k_i T \sum_{i=1}^{t} e_t + k_d \frac{e_t - e_{t-1}}{T}. \tag{6.2}$$

where k_p, k_i and k_d are gain constants for the corresponding term, and T is the sampling time. Fig. 6.3 shows the Simulink block diagram for the PID controller, where the gain constants are defined as $k_p = 9.4514, k_i = 0.69006, k_d = 2.8454$, and the sampling period is 0.2 s.

For the PID controller model, we check four different versions of implementations such as `PID1`, `PID2`, `PID3` and `PID3'`. `PID1` is obtained by code generation from the model using Embedded Coder. `PID2` is obtained from `PID1` by a manual transformation to improve the numerical accuracy (using the first transformation technique presented in [23]). In a similar way, `PID3` is obtained by the transformation from `PID1` for an even better numerical accuracy (following the optimization procedure and the output pseudo code of Listing 3 in [23]). However, the output code has an unintended bug by mistake that has been confirmed by the authors of the paper (i.e., variable `s` is not computed correctly, and the integral term is redundantly added to the output), which makes `PID3` incorrect. `PID3'` is an implementation that corrects `PID3`. Using LCV, we can verify that `PID1`, `PID2` and `PID3'` are correct implementations, but `PID3` is not (see Listing 6 in Appendix B).

Moreover, we check yet another version of implementation `PID4`. `PID4` is obtained by injecting a known bug of Embedded Coder into the implementation `PID1`. The bug with ID of 1658667 [71] that exists in the Embedded Coder version from 2015a through 2017b (7 consecutive versions) causes the generated code to have state variable declarations in a wrong scope. The state variables which are affected by the bug are mistakenly declared as local variables inside the step function instead of being declared as global variables. Thus, those state variables affected by the bug are unable to preserve their values throughout the consecutive step function executions. LCV can successfully detect the injected bug by identifying that the extracted model from the controller code does not match with the original controller model.

Quadrotor Controller

The second and more complex application in our case study is a controller of the quadrotor called Erle-Copter. The quadrotor controller controls the quadrotor to be in certain desired angles in roll, yaw and pitch. The quadrotor uses the controller software from the open source project Ardupilot[3]. Inspired by the controller software, we obtained the Simulink block diagram

[3]`http://ardupilot.org/`

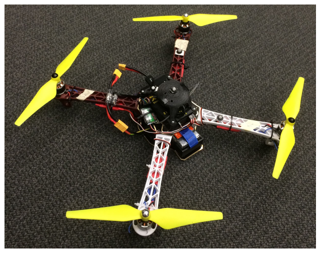
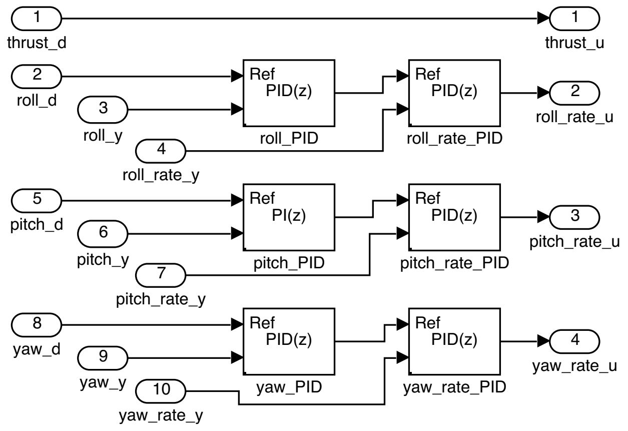

Figure 6.4: Our quadrotor platform (Left). The quadrotor controller block diagram (Right).

shown in Fig. 6.4. In the names of the inport blocks, the suffix `_d` indicates the desired angle, `_y`, the measured angle, and `_rate_y`, the angular speed. Each component of the coordinate of the quadrotor is separately controlled by its own cascade PID controller [56]. A cascade of PID controller is a sequential connection of two PID controllers such that one PID controller controls the reference point of another. In Fig. 6.4, there are three cascade controllers for the controls of roll, pitch and yaw. For example, for the roll control, `roll_pid` controls the angle of roll, while `roll_rate_PID` controls the rate of roll using the output of `roll_PID` as the reference point. The sampling time T of each PID controller is 2.5 ms. This model uses the built-in PID controller block of Simulink to enable the PID auto-tuning software in Matlab (i.e, `pidtune()`). The required physical quantities for controlling roll and pitch are identified by physical experiments [29]. We use Embedded Coder to generate the controller code for the model, and verify that the generated controller code correctly implements the controller model using LCV

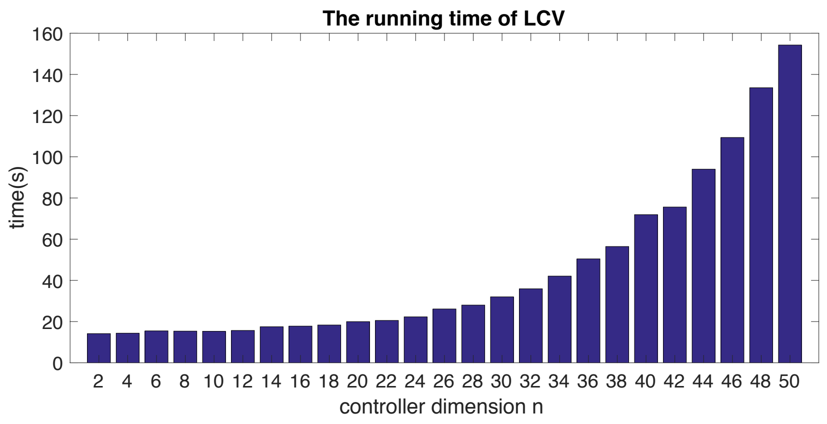

Figure 6.5: The running time of LCV for verifying controllers with dimension n.

(see Listing 7 in Appendix B).

6.2.2 Scalability

To evaluate the scalability of LCV with a larger range of controller dimension than what has been done in the previous chapters, we measure the running time of LCV verifying the controllers of different dimensions (i.e., the size of the LTI model) up to 50. We randomly generate LTI controller models using Matlab function `drss` varying the controller dimension n from 2 to 50. The range of controller sizes was chosen based on our observation of controller systems in practice. We construct Simulink models with LTI system blocks that contain the generated LTI models, and use Embedded Coder to generate the implementations for the controllers. The running time of LCV for verifying the controllers with different dimensions is presented in Fig. 6.5, which shows that LCV is scalable for the realistic size of controller dimension.

Chapter 7

Conclusion

7.1 Summary of this Dissertation

In conclusion, this dissertation addressed the problem of verifying linear controller software against its mathematical model in the absence of verified code generator in the model-based development. We developed an automatic verification tool to ensure the conformance between a step function C code and an LTI controller model from the input-output perspective with tolerance up to a given threshold. To develop LCV, we explored and proposed the approaches presented in Chapter 3 through Chapter 6.

In Chapter 3, we proposed to use invariants based on transfer functions, a well-known concept in the linear systems theory, since it allows us to accommodate 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 a 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.

In Chapter 4, we have proposed to use the symbolic execution technique to reconstruct mathematical models from linear time-invariant controller implementations. We have presented a method to check input-output equivalence between the specification model and the extracted model using the SMT formulation and the convex optimization formulation. Through the evaluation using randomly generated specification and code by Matlab, we showed that the scalability of our new approach has significantly improved compared to our own earlier work presented in Chapter 3.

In Chapter 5, we have presented an approach for the verification of finite precision implementations of linear controllers against mathematical specifications. We have proposed to use 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 presented in Chapter 4.

In Chapter 6, we have presented our tool LCV which verifies the equivalence between a given Simulink block diagram and a given C implementation from the input-output perspective. Through an evaluation, we have demonstrated that LCV is applicable to the verification of a real-world system's controller and scalable for the realistic controller size. We also demonstrated that LCV successfully detected certain known and unknown bugs of Embedded Coder and Salsa which are unverified code generation and transformation tools respectively which are used in the model-based development.

7.2 Future Research Direction

This dissertation has focused on linear controllers. A potential avenue of future work is extending the method/tool of this dissertation for non-linear controllers. This is motivated by the fact that Simuilnk provides a rich modeling language which is capable to specify many different types of non-LTI controllers. Although the class of LTI controllers are most commonly used in control systems, there are far more controller classes which are not LTI. However, we note that verification of certain types of non-linear controllers may not be decidable. Thus, an interesting extension of our verification approaches to a class of non-LTI controllers may be considering switched LTI controllers [50]. A switched LTI controller consists of multiple sub-controllers which are all LTI. These sub-controllers operate in parallel for the same input signal, but only one sub-controller's output is selected to be the actual control output to the actuator of the system. The selection of output signal is dome by a separate component called 'supervisor' (or decision maker). This class of controllers is a natural and interesting extension of LTI controllers that can model certain adaptive controllers. In the verification of this types of controllers implementations, we should be able to handle the 'Switch' block in the Simuilnk models and the conditional statement in the C language as well as the supervisor components which are potentially defined as state machines.

Another avenue of future work is finding the tolerance threshold values from robustness analysis. There is an inherent discrepancy between controller models and their implementation because not only do the implementations use finite-precision arithmetic, but they also may be inexact due to the potential rounding errors in the code generation/optimization process. Thus, it is reasonable to allow a tolerance in the conformance verification as long as the implementation has the same desired property to the model's. In our verification approaches presented in this thesis, we assumed that such a tolerance threshold value (e.g., approximate equivalence tolerance) is given by the control engineer as a result of robustness analysis. Thus, immediate future work may include the development of a technique to obtain a proper tolerance threshold value for the verification of a given controller building upon the existing robust control analysis techniques [32, 5, 82].

Appendix A

Step function example

```c
typedef double real_T;
typedef int int_T;
typedef char char_T;

typedef struct tag_RTM_LTIS_T RT_MODEL_LTIS_T;

typedef struct {
  real_T Internal_DSTATE[5];
} DW_LTIS_T;

typedef struct {
  real_T Internal_C[10];
} ConstP_LTIS_T;

typedef struct {
  real_T u[2];
} ExtU_LTIS_T;

typedef struct {
  real_T y[2];
} ExtY_LTIS_T;

struct tag_RTM_LTIS_T {
  const char_T * volatile errorStatus;
};

extern DW_LTIS_T LTIS_DW;
extern ExtU_LTIS_T LTIS_U;
extern ExtY_LTIS_T LTIS_Y;
extern const ConstP_LTIS_T LTIS_ConstP;
extern void LTIS_initialize(void);
extern void LTIS_step(void);
extern void LTIS_terminate(void);
extern RT_MODEL_LTIS_T *const LTIS_M;

const ConstP_LTIS_T LTIS_ConstP = {

  { -0.793176, 0.154365, -0.377883, -0.360608, -0.142123,
    0.503767, -0.573538, 0.170245, -0.583312, -0.56603 }
};
```

```
DW_LTIS_T LTIS_DW;
ExtU_LTIS_T LTIS_U;
ExtY_LTIS_T LTIS_Y;
RT_MODEL_LTIS_T LTIS_M_;
RT_MODEL_LTIS_T *const LTIS_M = &LTIS_M_;

void LTIS_step(void)
{
  {
    {
      static const int_T colCidxRow0[5] = { 0, 1, 2, 3, 4 };

      const int_T *pCidx = &colCidxRow0[0];
      const real_T *pC0 = LTIS_ConstP.Internal_C;
      const real_T *xd = &LTIS_DW.Internal_DSTATE[0];
      real_T *y0 = &LTIS_Y.y[0];
      int_T numNonZero = 4;
      *y0 = (*pC0++) * xd[*pCidx++];
      while (numNonZero--) {
        *y0 += (*pC0++) * xd[*pCidx++];
      }
    }

    {
      static const int_T colCidxRow1[5] = { 0, 1, 2, 3, 4 };

      const int_T *pCidx = &colCidxRow1[0];
      const real_T *pC5 = &LTIS_ConstP.Internal_C[5];
      const real_T *xd = &LTIS_DW.Internal_DSTATE[0];
      real_T *y1 = &LTIS_Y.y[1];
      int_T numNonZero = 4;
      *y1 = (*pC5++) * xd[*pCidx++];
      while (numNonZero--) {
        *y1 += (*pC5++) * xd[*pCidx++];
      }
    }
  }

  {
    real_T xnew[5];
    int_T i;
    xnew[0] = (0.87224)*LTIS_DW.Internal_DSTATE[0];
    xnew[0] += (0.822174)*LTIS_U.u[0]+(-0.438008)*LTIS_U.u[1];
    xnew[1] = (0.366378)*LTIS_DW.Internal_DSTATE[1];
    xnew[1] += (-0.278536)*LTIS_U.u[0]+(-0.824313)*LTIS_U.u[1];
    xnew[2] = (-0.540795)*LTIS_DW.Internal_DSTATE[2];
    xnew[2] += (0.874484)*LTIS_U.u[0]+(0.858857)*LTIS_U.u[1];
    xnew[3] = (-0.332664)*LTIS_DW.Internal_DSTATE[3];
    xnew[3] += (-0.117628)*LTIS_U.u[0]+(-0.506362)*LTIS_U.u[1];
    xnew[4] = (-0.204322)*LTIS_DW.Internal_DSTATE[4];
    xnew[4] += (-0.955459)*LTIS_U.u[0]+(-0.622498)*LTIS_U.u[1];

    for(i=0; i<5; i++)
        LTIS_DW.Internal_DSTATE[i] = xnew[i];
  }
}
```

Listing 5: Step function example

Appendix B

LCV output examples

```
------------------------
Verification parameters:
------------------------
Simulink model: dpid
Simulink block: dpid/dpid
C files:
    'dpid0.c'

Step function: dpid0_step
Input variables:
    'dpid0_U.r'    'dpid0_U.y'

Output variables:
    'dpid0_Y.u'

Precision(epsilon): 1e-05

---------------------
Verification started.
---------------------
Obtaining M1(A1,B1,C1,D1), a LTI model from the Simulink block diagram ...
A1
     1     0
     0     0


B1
   0.138012000000000  -0.138012000000000
  14.226999999999999 -14.226999999999999


C1
     1    -1


D1
  23.816411999999996 -23.816411999999996


Extracting M2(A2,B2,C2,D2), a LTI model from the C code ...
A2
     0     0     0     0     0     0     0
     1     0     0     0     0     0     0
     0     1     0     0     0     0     0
     0     0     1     0     0     0     0
     0     0     0     0     0     0     0
     0     0     0     0     0     1     0
     0     0     0     0     0     0     0


B2
   1.000000000000000  -1.000000000000000
                   0                   0
                   0                   0
                   0                   0
   1.000000000000000  -1.000000000000000
  14.227000000000000 -14.227000000000000
                   0                   0


C2
  Columns 1 through 5


  14.227000000000000  14.227000000000000  14.227000000000000  14.227000000000000  -0.138012000000000


  Columns 6 through 7
```

```
   1.000000000000000  14.227000000000000


D2
  38.043411999999996 -38.043411999999996


Checking the input-output equivalence ...
  Minimizing M1 as M1'(A1',B1',C1',D1') ...
  Minimizing M2 as M2'(A2',B2',C2',D2') ...
2 states removed.
Minimized models:
A1'
     1     0
     0     0


B1'
   0.138012000000000  -0.138012000000000
  14.226999999999999 -14.226999999999999


C1'
     1    -1


D1'
  23.816411999999996 -23.816411999999996


A2'
   0.000000000000000  -0.000000000000000   0.000000000000000   0.707106781186547   0.000000000000000
   1.000000000000000   0.000000000000000  -0.000000000000000  -0.000000000000000   0.000000000000000
   0.000000000000000   1.000000000000000   0.000000000000000   0.000000000000000  -0.000000000000000
   0.000000000000000  -0.000000000000000   0.000000000000000   0.000000000000000  -0.000000000000000
   0.000000000000000  -0.000000000000000  -0.000000000000000  -0.000000000000000   1.000000000000000


B2'
   0.000000000000000  -0.000000000000000
  -0.000000000000000   0.000000000000000
   0.000000000000000  -0.000000000000000
   1.414213562373095  -1.414213562373095
  14.227000000000000 -14.227000000000000
```

```
C2'
  14.227000000000000  14.227000000000000  14.227000000000000   9.962418954855893   1.000000000000000


D2'
  38.043411999999996 -38.043411999999996


Not equivalent (different dimension of minimized models) (2 ~= 5).
Elapsed time is 9.272016 seconds.


----------------------
Verification finished.
----------------------
```

Listing 6: Output of LCV for the PID3 example

```
------------------------
Verification parameters:
------------------------
Simulink model: erle_copter
Simulink block: erle_copter/erle_copter_controller
C files:
    'erle_copter_controller.c'

Step function: erle_copter_controller_step
Input variables:
  Columns 1 through 2

    'erle_copter_controller_U.thrust_d'    'erle_copter_controller_U.roll_d'

  Columns 3 through 4

    'erle_copter_controller_U.roll_y'    'erle_copter_controller_U.roll_rate_y'

  Columns 5 through 6

    'erle_copter_controller_U.pitch_d'    'erle_copter_controller_U.pitch_y'

  Columns 7 through 8

    'erle_copter_controller_U.pitch_rate_y'    'erle_copter_controller_U.yaw_d'

  Columns 9 through 10

    'erle_copter_controller_U.yaw_y'    'erle_copter_controller_U.yaw_rate_y'

Output variables:
  Columns 1 through 2

    'erle_copter_controller_Y.thrust_u'    'erle_copter_controller_Y.roll_rate_u'

  Columns 3 through 4

    'erle_copter_controller_Y.pitch_rate_u'    'erle_copter_controller_Y.yaw_rate_u'
```

```
Precision(epsilon): 1e-05

---------------------
Verification started.
---------------------
Obtaining M1(A1,B1,C1,D1), a LTI model from the Simulink block diagram ...
A1
   1.0e+02 *

  Columns 1 through 5

   0.010000000000000                   0                   0                   0                   0
                   0                   0                   0                   0                   0
   0.000000542625838  -0.000000542625838   0.010000000000000                   0                   0
   4.162826684542279  -4.162826684542278                   0                   0                   0
                   0                   0                   0                   0   0.010000000000000
                   0                   0                   0                   0   0.000000259717051
                   0                   0                   0                   0   4.160560411533799

  Columns 6 through 7

                   0                   0
                   0                   0
                   0                   0
                   0                   0
                   0                   0
   0.010000000000000                   0
                   0                   0

B1
   1.0e+05 *

  Columns 1 through 5

                   0   0.000000171758370  -0.000000171758370                   0                   0
                   0   0.006759018592030  -0.006759018592030                   0                   0
                   0   0.000000380868793  -0.000000380868793  -0.000000000542626                   0
```

```
                 0   2.921885886942280  -2.921885886942280  -0.004162826684542                   0
                 0                   0                   0                   0   0.000000000166662
                 0                   0                   0                   0   0.000000000302153
                 0                   0                   0                   0   0.004840360825814

  Columns 6 through 10

                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0
  -0.000000000166662                   0                   0                   0                   0
  -0.000000000302153  -0.000000000259717                   0                   0                   0
  -0.004840360825814  -0.004160560411534                   0                   0                   0

C1
   1.0e+02 *

  Columns 1 through 5

                 0                   0                   0                   0                   0
   4.177534754781552  -4.177534754781552   0.010000000000000  -0.010000000000000                   0
                 0                   0                   0                   0   4.170735794301745
                 0                   0                   0                   0                   0

  Columns 6 through 7

                 0                   0
                 0                   0
   0.010000000000000  -0.010000000000000
                 0                   0

D1
   1.0e+05 *

  Columns 1 through 5

   0.000010000000000                   0                   0                   0                   0
```

```
                 0   2.932209473801149  -2.932209473801149  -0.004177534754782                   0
                 0                   0                   0                   0   0.004852198780144
                 0                   0                   0                   0                   0

  Columns 6 through 10

                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0
  -0.004852198780144  -0.004170735794302                   0                   0                   0
                 0                   0   0.000010000000000  -0.000010000000000  -0.000010000000000

Extracting M2(A2,B2,C2,D2), a LTI model from the C code ...
A2
   1.0e+02 *

  Columns 1 through 5

                 0                   0                   0                   0                   0
                 0   0.010000000000000                   0                   0                   0
  -4.162826680000000   4.162826680000000                   0                   0                   0
  -0.000000540000000   0.000000540000000                   0   0.010000000000000                   0
                 0                   0                   0                   0   0.010000000000000
                 0                   0                   0                   0   4.160560410000000
                 0                   0                   0                   0   0.000000260000000
                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0

  Columns 6 through 10

                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0
                 0                   0                   0                   0                   0
                 0   0.010000000000000                   0                   0                   0
```

```
                  0                  0                  0                  0                  0
                  0                  0                  0   0.010000000000000                  0
                  0                  0                  0                  0                  0
                  0                  0                  0                  0                  0

  Column 11

                  0
                  0
                  0
                  0
                  0
                  0
                  0
                  0
                  0
                  0
   0.010000000000000

B2
   1.0e+05 *

  Columns 1 through 5

                  0   0.006759018590000  -0.006759018590000                  0                  0
                  0   0.000000171760000  -0.000000171760000                  0                  0
                  0   2.921885886960000  -2.921885886960000  -0.004162826680000                  0
                  0   0.000000380870000  -0.000000380870000  -0.000000000540000                  0
                  0                  0                  0                  0   0.000000000170000
                  0                  0                  0                  0   0.004840360830000
                  0                  0                  0                  0   0.000000000300000
                  0                  0                  0                  0                  0
                  0                  0                  0                  0                  0
                  0                  0                  0                  0                  0
                  0                  0                  0                  0                  0

  Columns 6 through 10
```

```
                   0                   0                   0                   0                   0
                   0                   0                   0                   0                   0
                   0                   0                   0                   0                   0
                   0                   0                   0                   0                   0
  -0.000000000170000                   0                   0                   0                   0
  -0.004840360830000  -0.004160560410000                   0                   0                   0
  -0.000000000300000  -0.000000000260000                   0                   0                   0
                   0                   0                   0                   0                   0
                   0                   0                   0                   0                   0
                   0                   0                   0                   0                   0
                   0                   0                   0                   0                   0


C2

   1.0e+02 *


  Columns 1 through 5


                   0                   0                   0                   0                   0
  -4.177534750000000   4.177534750000000  -0.010000000000000   0.010000000000000                   0
                   0                   0                   0                   0   4.170735790000000
                   0                   0                   0                   0                   0


  Columns 6 through 10


                   0                   0                   0                   0                   0
                   0                   0                   0                   0                   0
  -0.010000000000000   0.010000000000000                   0                   0                   0
                   0                   0  -0.010000000000000   0.010000000000000  -0.010000000000000


  Column 11


                   0
                   0
                   0
   0.010000000000000


D2

   1.0e+05 *
  Columns 1 through 5

   0.000010000000000                  0                  0                  0                  0
                   0   2.932209473820000  -2.932209473820000  -0.004177534750000                  0
                   0                  0                  0                  0   0.004852198780000
                   0                  0                  0                  0                  0

  Columns 6 through 10

                   0                  0                  0                  0                  0
                   0                  0                  0                  0                  0
  -0.004852198780000  -0.004170735790000                  0                  0                  0
                   0                  0   0.000010000000000  -0.000010000000000  -0.000010000000000

Checking the input-output equivalence ...
  Minimizing M1 as M1'(A1',B1',C1',D1') ...
4 states removed.
  Minimizing M2 as M2'(A2',B2',C2',D2') ...
8 states removed.
Minimized models:
A1'
   1.0e+02 *

   0.000000553104392   0.000000000000110  -0.000000000000072
  -0.053294848472191  -0.000000010572831   0.000000006946340
   4.162485496370146   0.000000825769400  -0.000000542529747

B1'
   1.0e+05 *

  Columns 1 through 5

                   0  -0.006758630368344   0.006758630368344  -0.000000000553104  -0.000000000952093
                   0  -0.037407627962002   0.037407627962002   0.000053294848714   0.004839964128421
                   0   2.921646421271873  -2.921646421271873  -0.004162485515247   0.000061969021889

  Columns 6 through 10
```

```
   0.000000000952093   0.000000000818377                   0                   0                   0
  -0.004839964128421  -0.004160219427973                   0                   0                   0
  -0.000061969021889  -0.000053265834612                   0                   0                   0

C1'
   1.0e+02 *

                   0                   0                   0
   4.177534734505039   0.000128854363016  -0.009999723626231
   0.000000001966988  -0.009999180438304  -0.000128025624779
                   0                   0                   0

D1'
   1.0e+05 *

  Columns 1 through 5

   0.000010000000000                   0                   0                   0                   0
                   0   2.932209473801149  -2.932209473801149  -0.004177534754782                   0
                   0                   0                   0                   0   0.004852198780144
                   0                   0                   0                   0                   0

  Columns 6 through 10

                   0                   0                   0                   0                   0
                   0                   0                   0                   0                   0
  -0.004852198780144  -0.004170735794302                   0                   0                   0
                   0                   0   0.000010000000000  -0.000010000000000  -0.000010000000000

A2'
   1.0e+02 *

   0.000000000033993   0.000446804329864   0.000000000057955
  -0.000000000000041  -0.000000539991988  -0.000000000000070
   0.000000315643503   4.162826637142502   0.000000539959925

B2'
```

```
   1.0e+05 *

  Columns 1 through 5

                  0   0.000313611216248  -0.000313611216248  -0.000000446804332   0.004840360802119
                  0  -0.006759397610099   0.006759397610099   0.000000000539992  -0.000000000366950
                  0   2.921885869253011  -2.921885869253011  -0.004162826656022  -0.000000519525398

  Columns 6 through 10

  -0.004840360802119  -0.004160560386035                   0                   0                   0
   0.000000000366950   0.000000000315414                   0                   0                   0
   0.000000519525398   0.000000446561089                   0                   0                   0

C2'
   1.0e+02 *

                  0                   0                   0
  -0.000000756560691   4.177534732347563  -0.009999456777501
  -0.009999999942204   0.000000000758104   0.000001073319564
                  0                   0                   0

D2'
   1.0e+05 *

  Columns 1 through 5

   0.000010000000000                   0                   0                   0                   0
                  0   2.932209473820000  -2.932209473820000  -0.004177534750000                   0
                  0                   0                   0                   0   0.004852198780000
                  0                   0                   0                   0                   0

  Columns 6 through 10

                  0                   0                   0                   0                   0
                  0                   0                   0                   0                   0
  -0.004852198780000  -0.004170735790000                   0                   0                   0
                  0                   0   0.000010000000000  -0.000010000000000  -0.000010000000000
```

```
M1' and M2' are input-output equivalent.
Similarity transformation matrix T found with error bound 4.2483e-06:
  -0.000000120860229   0.999999999995001   0.000000262597397
   0.999916663315123   0.000000118167540  -0.012909885578054
   0.012909885557525   0.000000213285385   0.999916665046047

Elapsed time is 14.326160 seconds.
----------------------
Verification finished.
----------------------
```

Listing 7: Output of LCV for the Erle Copter example

Bibliography

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

[2] IEEE standard for floating-point arithmetic. IEEE Std 754-2008 pp. 1–70, 2008.

[3] PolyFP. https://github.com/monadius/poly_fp, accessed 2016.

[4] Tony Andrews, Shaz Qadeer, Sriram Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *Computer Aided Verification*, pages 28–32. Springer, 2004.

[5] Adolfo Anta, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. Automatic verification of control system implementations. In *Proc. 10th ACM International Conference on Embedded Software*, EMSOFT'10, pages 9–18, 2010.

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

[7] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, et al. The Coq proof assistant reference manual: Version 6.1. 1997.

[8] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[9] Patrick Baudin, Franois Bobot, Loc Correnson, and Zaynah Dargaye. WP 0.8 manual - Frama-C. Technical report, CEA LIST, 2014.

[10] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliatre, Claude Marche, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language, Version 1.4. Technical report, CEA LIST and INRIA, 2010.

[11] Chandan Kumar Behera and D Lalitha Bhaskari. Different obfuscation techniques for code protection. *Procedia Computer Science*, 70:757–763, 2015.

[12] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN Notices*, volume 38, pages 196–207. ACM, 2003.

[13] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[14] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121, 2006.

[15] Jan Cappaert. Code obfuscation techniques for software protection. *Katholieke Universiteit Leuven*, pages 1–112, 2012.

[16] Nuno Carvalho, Cristiano da Silva Sousa, Jorge Sousa Pinto, and Aaron Tomb. Formal Verification of kLIBC with the WP Frama-C Plug-in. In *NASA Formal Methods*, pages 343–358. Springer, 2014.

[17] Lori Clarke. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, (3):215–222, 1976.

[18] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[19] Mirko Conrad. Testing-based translation validation of generated code in the context of iec 61508. *Formal Methods in System Design*, 35(3):389–401, 2009.

[20] Mirko Conrad. Verification and validation according to iso 26262: A workflow to facilitate the development of high-integrity software. *Embedded Real Time Software and Systems (ERTS2 2012)*, 2012.

[21] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Păsăreanu, Ro Bby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2000.

[22] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *Software Engineering and Formal Methods*, pages 233–247. 2012.

[23] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. Transformation of a pid controller for numerical accuracy. *Electronic Notes in Theoretical Computer Science*, 317:47–54, 2015.

[24] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. Improving the numerical accuracy of programs by automatic transformation. *International Journal on Software Tools for Technology Transfer*, pages 1–22, 2016.

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

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

[27] Leonardo De Moura, Sam Owre, Harald Rueß, John Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. Sal 2. In *International Conference on Computer Aided Verification*, pages 496–500. Springer, 2004.

[28] Mickaël Delahaye, Nikolai Kosmatov, and Julien 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.

[29] L Derafa, T Madani, and A Benallegue. Dynamic modelling and experimental identification of four rotors helicopter parameters. In *2006 IEEE International Conference on Industrial Technology*, 2006.

[30] Edsger Wybe Dijkstra. *A discipline of programming*. Prentice-Hall Englewood Cliffs, 1976.

[31] Geir Dullerud and Fernando Paganini. *Course in Robust Control Theory*. Springer-Verlag New York, 2000.

[32] Geir E Dullerud and Fernando Paganini. *A course in robust control theory: a convex approach*, volume 36. Springer Science & Business Media, 2013.

[33] Hassan Eldib and Chao 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.

[34] Jérôme Feret. Static analysis of digital filters. In *European Symposium on Programming*, pages 33–48. Springer, 2004.

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

[36] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5):397–403, 2011.

[37] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *ICFEM*, volume 3308, pages 15–29. Springer, 2004.

[38] Frédéric Goualard. How do you compute the midpoint of an interval? *ACM Transactions on Mathematical Software (TOMS)*, 40(2):11, 2014.

[39] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 232–247. Springer, 2011.

[40] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx`, March 2014.

[41] Kerstin Hartig, Jens Gerlach, Juan Soto, and Jürgen 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.

[42] Heber Herencia-Zapana, Romain Jobredeaux, Sam Owre, Pierre-Loïc Garoche, Eric Feron, Gilberto Perez, and Pablo Ascariz. PVS linear algebra libraries for verification of control software algorithms in C/ACSL. In *NASA Formal Methods*, pages 147–161. 2012.

[43] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10):576–580, 1969.

[44] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[45] Gerard J Holzmann and Margaret H Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification and Reliability*, 11(2):65–79, 2001.

[46] Gerard J Holzmann and Margaret H Smith. An automated verification method for distributed systems software based on model extraction. *Software Engineering, IEEE Transactions on*, 28(4):364–377, 2002.

[47] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

[49] David Kung, Nimish Suchak, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, and Chris Chen. On object state testing. In *Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International*, pages 222–227. IEEE, 1994.

[50] Daniel Liberzon and A Stephen Morse. Basic problems in stability and design of switched systems. *IEEE Control systems*, 19(5):59–70, 1999.

[51] Rupak Majumdar, Indranil Saha, KC Shashidhar, and Zilong Wang. CLSE: Closed-loop symbolic execution. In *NASA Formal Methods*, pages 356–370. 2012.

[52] Rupak Majumdar, Indranil Saha, Koichi Ueda, and Hakan 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.

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

[54] Tomas Matousek and Filip Zavoral. Extracting Zing models from C source code. *SOFSEM 2007: Theory and Practice of Computer Science*, pages 900–910, 2007.

[55] Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[56] Nathan Michael, Daniel Mellinger, Quentin Lindsey, and Vijay Kumar. The grasp multiple micro-uav test bed. *IEEE Robotics & Automation Magazine*, 17(3):56–65, 2010.

[57] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[58] Miroslav Pajic, Junkil Park, Insup Lee, George J Pappas, and Oleg Sokolsky. Automatic verification of linear controller software. In *12th International Conference on Embedded Software (EMSOFT)*, pages 217–226. IEEE Press, 2015.

[59] Junkil Park. Step function example.

[60] Junkil Park, Miroslav Pajic, Insup Lee, and Oleg Sokolsky. Scalable verification of linear controller software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 662–679. Springer, 2016.

[61] Junkil Park, Miroslav Pajic, Oleg Sokolsky, and Insup Lee. Automatic verification of finite precision implementations of linear controllers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 153–169. Springer, 2017.

[62] Josef Pichler. Specification extraction by symbolic execution. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 462–466. IEEE, 2013.

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

[64] Wilson J Rugh. *Linear system theory.* Prentice Hall, 1996.

[65] Michael Ryabtsev and Ofer Strichman. Translation validation: From simulink to c. In *Computer Aided Verification*, pages 696–701. Springer, 2009.

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

[67] Tamal Sen and Rajib Mall. Extracting finite state representation of Java programs. *Software & Systems Modeling*, 15(2):497–511, 2016.

[68] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *International Symposium on Formal Methods*, pages 532–550. Springer, 2015.

[69] Stéphane S Somé and Timothy C Lethbridge. Enhancing program comprehension with recovered state models. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 85–93. IEEE, 2002.

[70] Ingo Stuermer, Mirko Conrad, Heiko Doerr, and Peter Pepper. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9):622–634, 2007.

[71] The Mathworks, Inc. Bug Reports for Incorrect Code Generation. `http://www.mathworks.com/support/bugreports/?product=ALL&release=R2015b&keyword=Incorrect+Code+Generation`.

[72] The Mathworks, Inc. Embedded coder. `https://www.mathworks.com/products/embedded-coder.html`, September 2017.

[73] The Mathworks, Inc. Simulink. `https://www.mathworks.com/products/simulink.html`, September 2017.

[74] The Mathworks, Inc. Simulink coder. `https://www.mathworks.com/products/simulink-coder.html`, September 2017.

[75] The Mathworks, Inc. Simulink design verifier. `https://www.mathworks.com/products/sldesignverifier.html`, September 2017.

[76] The Mathworks, Inc. Simulink test. `https://www.mathworks.com/products/simulink-test.html`, September 2017.

[77] The Mathworks, Inc. Stateflow. `https://www.mathworks.com/products/stateflow.html`, September 2017.

[78] Sean Thompson. A survey on model checking Java programs. Technical report, Technical Report CSRG-407. Department of Computer Science, University of Toronto, 2000.

[79] Shaohui Wang, Srinivasan Dwarakanathan, Oleg Sokolsky, and Insup Lee. High-level model extraction via symbolic execution. Technical Reports (CIS) Paper 967, University of Pennsylvania, http://repository.upenn.edu/cis_reports/967, 2012.

[80] Timothy Wang, Romain Jobredeaux, Heber Herencia, Pierre-Loic Garoche, Arnaud Dieumegard, Eric Feron, and Marc Pantel. From design to implementation: an automated, credible autocoding chain for control systems. *arXiv preprint arXiv:1307.2641*, 2013.

[81] Timothy E Wang, Alireza Esna Ashari, Romain J Jobredeaux, and Eric M Feron. Credible autocoding of fault detection observers. In *American Control Conference (ACC)*, pages 672–677, 2014.

[82] Timothy E Wang, Pierre-Loïc Garoche, Pierre Roux, Romain Jobredeaux, and Éric Féron. Formal analysis of robustness at model and code level. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, pages 125–134. ACM, 2016.

[83] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing-EDCC 5*, pages 281–292. Springer, 2005.