Automatic Verification of Control System Implementations*

Adolfo Anta[1], Rupak Majumdar[1,2], Indranil Saha[1], Paulo Tabuada[1]
[1]University of California Los Angeles
[2]MPI-SWS
{adolfo, tabuada}@ee.ucla.edu, {rupak, indranil}@cs.ucla.edu

ABSTRACT

Software implementations of controllers for physical subsystems form the core of many modern safety-critical systems such as aircraft flight control and automotive engine control. A fundamental property of such implementations is *stability*, the guarantee that the physical plant converges to a desired behavior under the actions of the controller. We present a methodology and a tool to perform automated static analysis of embedded controller code for stability of the controlled physical system.

The design of controllers for physical systems provides not only the controllers but also mathematical proofs of their stability under idealized mathematical models. Unfortunately, since these models do not capture most of the implementation details, it is not always clear if the stability properties are retained by the software implementation, either because of software bugs, or because of imprecisions arising from fixed-precision arithmetic or timing.

Our methodology is based on the following separation of concerns. First, we analyze the controller mathematical models to derive bounds on the implementation errors that can be tolerated while still guaranteeing stability. Second, we automatically analyze the controller software to check if the maximal implementation error is within the tolerance bound computed in the first step.

We have implemented this methodology in Costan, a tool to check stability for controller implementations. Using Costan, we analyzed a set of control examples whose mathematical models are given in Matlab/Simulink and whose C implementation is generated using Real-Time Workshop. Unlike previous static analysis research, which has focused on proving low-level runtime properties such as absence of buffer overruns or arithmetic overflows, our technique combines analysis of the mathematical controller models and automated analysis of source code to guarantee application-level stability properties.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software Engineering—*Program Verification*

*This research was funded in part by the NSF grants CNS-0720881 and CCF-0953994, the DARPA grant HR0011-09-1-0037, and a grant from Toyota Corporation.

General Terms

Verification

Keywords

Controller Design, Stability, Program Analysis, Fixed-point Arithmetic

1. INTRODUCTION

Software implementations of controllers for dynamical systems play a key role in many safety-critical domains. One of the most important properties of such controllers is *stability*, the requirement that the controlled system remains close to some reference behavior (and ultimately, converges to the reference behavior).

The theory of control of dynamical systems provides a mathematical basis for stability analysis of controlled systems. A model of the dynamical system to be controlled (e.g., an engine or a chemical plant) is constructed, either as a system of ordinary differential equations or difference equations and a *feedback controller* is designed to provide control inputs to the system based on observations of the system state. The model and the controller are then analyzed to ensure that the system is stable in the desired operating region. If stable, the controller is implemented in software or hardware. The process is aided by software tools that provide mathematical modeling, simulation, and automatic code generation.

Unfortunately, there is a large semantic gap between the mathematical analysis of dynamical systems and controller implementations in software. Whereas the control engineer proves stability of the system *at the continuous, mathematical level*, the controller is implemented in software with discrete sensing and actuation of physical signals, limited precision arithmetic, resource contention, and potentially faulty sensors and actuators. Thus, it is often unclear if the mathematical proof of stability continues to hold for the implemented system. The current practice is to perform extensive simulation of the generated code to gain confidence that the implementation continues to satisfy the stability properties proved for the mathematical model. While simulation helps weed out bugs, it is usually non-exhaustive, and thus cannot be used to argue correctness. For safety-critical systems, it is desirable to develop techniques that can ensure correctness of the implementation artifact.

In this paper, we introduce a methodology for the design and automatic verification of control system implementations for stability properties. Our methodology has two components. First, we provide a mathematical analysis of stability that incorporates implementation errors arising out of the mismatch between the ideal mathematical controller and its software implementation. Our analysis provides a relationship between the region where the continuous variables can be steered to under the action of the controller and the error between the ideal controller and its implementation.

Second, we perform static program analysis on the software code implementing the controller to compute an upper bound on the error between the mathematical control signal and the software output. The static analysis is based on verification condition generation [23], and reduces the error bound question to a validity problem for a formula in the combination theory of reals and bitvectors, for which off-the-shelf, efficient decision procedures are available [16]. Together with the first part, this enables us to compute an upper bound on the region to which the *implemented* controller is guaranteed to steer the system.

We have implemented Costan (for Controller Stability Analyzer), a tool that reads in controller implementations in C and checks the maximum possible error between the outputs of the C implementation and the Simulink function implementing the mathematical controller. The current version of Costan targets *fixed-point* implementations of controllers. Fixed-point implementations use fixed width (e.g., 16- or 32-bit) integers to implement numerical operations over real numbers. They are useful for implementing controllers on hardware platforms without floating point support. (Our methodology carries over to floating point implementations, given the semantics of floating point operations in the verification condition generation [6, 12].) Using Costan, we have analyzed error bounds for a number of linear and non-linear control systems implementations, including a realistic train car controller [21], and we have characterized the regions to which the controllers can steer the physical variables.

Control systems code provides a sweet spot for static checkers: while the application domain is often safety-critical —making verification desirable— the code itself usually has statically unrollable loops and statically allocated memory, removing language features that are the bane of most static analyzers. Moreover, unlike generic software, software for control systems often comes with mathematical models and specifications. This allows a co-ordinated analysis by pushing some of the complexity of software analysis to the model level (e.g., without the mathematical analysis and the plant model, it is difficult, even impossible, to judge whether the numerical manipulations of the controller give rise to stable behaviors [18]). Our results and implementation show how domain knowledge from control theory on the one hand and static program verification techniques on the other can interact to provide a solution to an important reliability problem.

Related Work. While there has been a lot of work on static analysis of safety-critical control system implementation code [4, 13, 9, 17, 5, 12, 10], the main emphasis in previous work has been on low-level properties such as arithmetic overflows or buffer overruns. By incorporating mathematical analysis of control design into our methodology, we can focus on *application level* properties such as stability. By putting the control designer "in the loop" we can move the complexity of some of this analysis from the static analysis tool to the mathematics of control. While we describe a program analysis based on verification-condition generation, analyses based on abstract interpretation, such as Astree [4] and Fluctuat [17, 5] will also be applicable to the analysis.

An alternate approach outlined in [14, 15, 1] generates mathematical stability proofs and *compiles* such proofs into the implementation. We believe our methodology holds the advantage of *separation of concerns*. By forcing the implementation to conform to one of several possible stability proofs, we limit the space of implementation and optimization options. Instead, by producing a —more abstract— relationship between implementation errors and the regions where the physical variables can be steered to, we are free to explore different implementations. For example, in our experiments, we consider a non-linear control system with two different implementations: one based on a direct evaluation of a polynomial control law and a second based on the evaluation of the same polynomial using a lookup table and interpolation. Compiling the stability proof for the second option (which looks very different from the mathematical polynomial function) would be hard. Further, the compilation of stability proofs requires extensive changes to already complex auto-code generators to produce the proof of stability along with the implementation. Moreover, if controller implementations are written from scratch, the compilation strategy does not work, but our methodology, being agnostic to the source of the controller, does.

2. MOTIVATING EXAMPLE

We start with a simple example that shows the controller design process and describes the semantic gap between a mathematical controller and its implementation. We consider a simple physical model of a locomotive pulling a train car where the connection between the locomotive and the car is modeled by a spring in parallel with a damper. The model has been borrowed from [21]. The dynamics of this system is given by:

$$\frac{d}{dt}\begin{bmatrix}\xi_1\\ \xi_2\\ \xi_3\end{bmatrix} = \begin{bmatrix}0 & 1 & -1\\ -\frac{k}{m_1} & -\frac{b+c}{m_1} & \frac{b}{m_1}\\ \frac{k}{m_2} & \frac{b}{m_2} & -\frac{b}{m_2}\end{bmatrix}\begin{bmatrix}\xi_1\\ \xi_2\\ \xi_3\end{bmatrix} + \begin{bmatrix}0\\ \frac{1}{m_1}\\ 0\end{bmatrix}\upsilon \quad (1)$$

$$\omega = \begin{bmatrix}0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\xi_1\\ \xi_2\\ \xi_3\end{bmatrix} \quad (2)$$

where ξ_1 represents the distance between the locomotive and the car, ξ_2 and ξ_3 represent the velocities of the locomotive and the car, $m_1 = 176580$Kg is the mass of the locomotive, $m_2 = 100698$Kg is the mass of the car, $b = 1100$Ns/m is the damping coefficient, $k = 7874$N/m is the restitution coefficient of the spring, $c = 160$Ns/m is the aerodynamic friction coefficient, and υ represents the force applied by the locomotive.

The control objective is to quickly reduce the possible oscillations between the locomotive and the car. Through a simple change of coordinates this problem can be reformulated to the design of a controller that forces ξ_1 to converge to zero. One possible state feedback controller for the system is given by:

$$\upsilon = F\xi, \qquad F = [-0.1763 \quad -0.3494 \quad 0.0602]\,. \quad (3)$$

However, if only the velocities of the locomotive and car can be measured, (3) cannot be directly implemented. Hence, the feedback controller (3) is extended with an observer estimating the distance between the locomotive and the car as follows:

$$\frac{d}{dt}\begin{bmatrix}\zeta_1\\ \zeta_2\\ \zeta_3\end{bmatrix} = \begin{bmatrix}0 & 79.58 & -127.95\\ -1.04 & -10.68 & -1.19\\ 0.07 & 1.49 & -11.26\end{bmatrix}\begin{bmatrix}\zeta_1\\ \zeta_2\\ \zeta_3\end{bmatrix} + \begin{bmatrix}-78.58 & 126.95\\ 8.69 & 1.54\\ -1.48 & 11.25\end{bmatrix}\omega. \quad (4)$$

The resulting controller consists of the estimator and the feedback control law (3) evaluated at the estimate ζ, *i.e.*:

$$\upsilon = F\zeta.$$

Figure 2 shows (part of the) fixed-point implementation of the controller. The code was generated automatically by Real Time Workshop. We point out two observations about the implementation. First, the controller is implemented using finite precision, fixed-point numbers, and "looks" very different from the mathematical controller. For example, the last five lines of the controller code implement the matrix multiplication (3), but the correspondence is not obvious from the code. Second, the output of the implementation need not exactly match the mathematical control function, due to finite precision arithmetic. In particular, it is not

```
 int DelayStateX_DSTATE[3];   // fixed point type: signed, 32bits, 28 fractional bits
 int Xk[3], CXk[3];           // fixed point type: signed, 32bits, 28 fractional bits
 int Gain1;                   // fixed point type: signed, 32bits, 29 fractional bits
 int Gain2;                   // fixed point type: signed, 32bits, 28 fractional bits
 int u1;                      // fixed point type: signed, 32bits, 28 fractional bits

static void controller(void)
{
  int tmp, tmp0;
  Xk[0] = DelayStateX_DSTATE[0]; Xk[1] = DelayStateX_DSTATE[1]; Xk[2] = DelayStateX_DSTATE[2];
  Gain2 = 310689525;  // Reference input

  ...  // update observer state

  // compute control output based on state and observation
  for (tmp = 0; tmp < 3; tmp++) {
    tmp0 = (int)((long int)OutputMatrixC_Gain[tmp] * (longint)Xk[0] ≫ 30U);
    tmp0 += (int)((long int)OutputMatrixC_Gain[tmp + 3] * (long int)Xk[1] ≫ 30U);
    tmp0 += (int)((long int)OutputMatrixC_Gain[tmp + 6] * (long int)Xk[2] ≫ 30U);
    CXk[tmp] = tmp0;
  }
  tmp = (int)((long int)(-757257017) * (long int)CXk[0] ≫ 31U);
  tmp += (int)((long int)(-1500825839) * (long int)CXk[1] ≫ 31U);
  tmp += (int)((long int)258754934 * (long int)CXk[2] ≫ 31U);
  Gain1 = tmp;
  u1 = (Gain1 ≫ 1) + Gain2;
}
```

Figure 1: Fixed point controller implementation for locomotive and train car (generated by Fixed Point Advisor and Real Time Workshop)

clear if this C implementation maintains the stability properties that were guaranteed by the mathematical design.

In the rest of the paper, we describe our methodology that enables us to reason about the mathematical control system in conjunction with the implemented code. In Section 3, we describe an analysis that quantifies the effect of implementation errors on system stability. In Section 4, we describe how the implementation errors can be computed from the C implementation. These two analyses together enable us to ascertain stability properties of the controller implementation.

3. MATHEMATICAL DESIGN OF CONTROLLERS

3.1 Notation

We denote by $\mathbb{R}$, $\mathbb{R}_0^+$, and $\mathbb{R}^+$ the real, non-negative real, and positive real numbers, respectively. The natural numbers, including zero, are denoted by $\mathbb{N}_0$. The function e^{At}, for $t \in \mathbb{R}$ denotes the matrix function defined by the convergent series:

$$\mathsf{e}^{At} = 1_{n\times n} + At + \frac{1}{2!}A^2t^2 + \frac{1}{3!}A^3t^3 + \dots$$

where $1_{n\times n}$ is the identity matrix with n rows and n columns and e is Euler's constant. The zero matrix with n rows and m columns is denoted by $0_{n\times m}$. The j-th entry of a vector $x \in \mathbb{R}^n$ is denoted by x_j and the Euclidean norm of x is given by $\|x\| = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$. We will also use the 1-norm, denoted by $\|x\|_1$ and defined by $\|x\|_1 = |x_1|+|x_2|+\dots+|x_n|$, where $|x_j|$ is the absolute value of $x_j \in \mathbb{R}$. These two norms are related by $\|x\| \le \|x\|_1 \le \sqrt{n}\|x\|$. The induced norm of the matrix P is defined by $\|P\| = \max_{\|x\|=1} \|Px\|$.

3.2 Differential equations, controllers, and observers

Physical systems are typically modeled by differential equations:

$$\frac{d}{dt}\xi = f(\xi, \upsilon) \qquad (5)$$

in which the curve $\xi : \mathbb{R} \to \mathbb{R}^n$ describes how the physical quantities of interest change over time. At each time instant $t \in \mathbb{R}$, $\xi(t)$ is a vector in $\mathbb{R}^n$ containing the values of physical quantities such as positions, velocities, temperatures, pressures, etc. The curve ξ is said to be a *solution* or *trajectory* of (5) when there exists an input curve $\upsilon : \mathbb{R} \to \mathbb{R}^m$ such that the time derivative of ξ at time t equals $f(\xi(t), \upsilon(t))$. We denote by $\xi_{x\upsilon}(t)$ the point reached by the solution ξ of (5) at time $t \in \mathbb{R}$, starting at the state $x \in \mathbb{R}^n$ and under the input $\upsilon : \mathbb{R} \to \mathbb{R}^m$. Note that different input curves υ give rise to different trajectories ξ. The objective of the control engineer is to design a controller $\upsilon = k(\xi, c)$ computing υ based on the evolution of the physical variables and the desired commands c given by the user. The resulting input curves υ force the corresponding trajectories ξ to have desirable properties, most notably *asymptotic stability.* Intuitively, given a desired operating point $x_0 \in \mathbb{R}^n$, we would like to design input curves υ so that trajectories ξ of (5) converge asymptotically to x_0, *i.e.,* $\lim_{t\to\infty} \xi(t) = x_0$. Moreover, stability also requires that if $\xi(0)$ is close to x_0, then $\xi(t)$ should remain close to x_0 for $t > 0$. Whenever the controller $\upsilon = k(\xi, c)$ gives rise to trajectories with the preceding properties, we say that x_0 is a global asymptotically stable equilibrium point for

$$\frac{d}{dt}\xi = f(\xi, k(\xi, c)).$$

For simplicity, we restrict attention to *linear* differential equations:

$$\frac{d}{dt}\xi = A\xi + B\upsilon, \qquad \xi(t) \in \mathbb{R}^n, \upsilon(t) \in \mathbb{R}^m \qquad (6)$$

$$\omega = C\xi, \qquad \omega(t) \in \mathbb{R}^p \qquad (7)$$

where the letters A, B, and C denote matrices of appropriate dimensions. An example of a linear differential equation is the model of the train given by (1) and (2) in Section 2. In Section 5 we shall outline how the work described in this paper extends to the nonlinear case. We will also focus our attention on the operation point $x_0 = 0$ without loss of generality.

The states $x \in \mathbb{R}^n$ can be measured by equipping the physical system with sensors. Unfortunately, either for economic considerations or for technical difficulties, it is usually not possible to measure the complete state $x \in \mathbb{R}^n$ and only the output $Cx = o \in \mathbb{R}^p$ $(p < n)$ can be obtained from the sensors. Nonetheless, most of the existing design methodologies for controllers assume that the full state is available for control purposes. This mismatch between the available information and the desired information about the state is handled by first designing a linear controller:

$$\upsilon = F\xi \tag{8}$$

making $x_0 = 0$ an asymptotically stable equilibrium point of:

$$\frac{d}{dt}\xi = A\xi + BF\xi.$$

Note that controller (8) assumes that the full state is available. Since this is not the case, (8) is evaluated at the estimate ζ of the state evolution provided by an observer:

$$\begin{aligned} \frac{d}{dt}\zeta &= D\zeta + E\omega, \qquad \zeta(t) \in \mathbb{R}^n & (9)\\ \upsilon &= F\zeta. & (10) \end{aligned}$$

The matrices D and E are designed so as to ensure that the estimate ζ converges to ξ. A well known result in control theory, the *separation principle* [2], is then invoked to show that by evaluating the controller on the estimate ζ, stability is still guaranteed. For the train example in Section 2 the observer is given by (4) while the linear controller is described by (3).

3.3 Implementation Errors

The controller design and analysis described in the previous section is done using continuous-time mathematical models that fail to describe the details of the controller implementation such as digital sampling and finite precision arithmetic. In order to describe the mismatch between the controller specification (9) and (10), and its software implementation, we first consider the exact discrete-time version of (9) and (10):

$$\begin{aligned} z[r+1] &= D_T z[r] + E_T o[r] + e_s & (11)\\ u[r+1] &= F z[r+1], \end{aligned}$$

where the matrices D_T and E_T are given by:

$$D_T = \mathrm{e}^{DT}, \qquad E_T = \int_{rT}^{(r+1)T} \mathrm{e}^{D(T-\tau)} E d\tau,$$

and T is the sampling period. The signals z and o describe the exact value of the signals ζ and ω at the discrete-time sampling instants $0, T, 2T, 3T, \ldots$. Mathematically, we have:

$$z[r] = \zeta(rT), \qquad o[r] = \omega(rT), \quad \forall r \in \mathbb{N}_0.$$

The term e_s is the sampling error describing the mismatch between the continuous-time signal ω and the discrete-time signal o:

$$\begin{aligned} e_s &= \int_{rT}^{(r+1)T} \mathrm{e}^{D(T-\tau)} E \varepsilon_s(\tau) d\tau \\ &= \int_{rT}^{(r+1)T} \mathrm{e}^{D(T-\tau)} E \left(\omega(\tau) - o[r]\right) d\tau \end{aligned}$$

It can be shown that by sampling sufficiently fast, the error e_s can be rendered arbitrarily small. Since typical embedded controller implementations use periods in the millisecond to microsecond range, we will make this fast sampling assumption to conclude that quantization errors dominate the sampling errors. Hence, in the following discussion we will assume that $\varepsilon_s = 0$.

A second source of errors is quantization due to the fixed-point implementation of the controller code. We use the notation $Q(x)$ and $Q(P)$ to denote the fixed-point representation of a vector $x \in \mathbb{R}^n$ and a matrix $P \in \mathbb{R}^{n \times m}$, respectively. A possible software implementation of the controller (9) and (10) can then be described by:

$$\begin{aligned} \widehat{z}[r+1] &= Q\left(Q(Q(D_T)\widehat{z}[r]) + Q(Q(E_T)Q(o[r]))\right) \\ &= D_T\widehat{z}[r] + E_T o[r] + e_{q1} \\ \widehat{u}[r+1] &= Q\left(Q(F)\widehat{z}[r+1]\right) \\ &= F\widehat{z}[r+1] + e_{q2}. & (12) \end{aligned}$$

where

$$\begin{aligned} e_{q1} &= Q\left(Q(Q(D_T)\widehat{z}) + Q(Q(E_T)Q(o))\right) \\ &\quad -D_T\widehat{z} - E_T o \\ e_{q2} &= Q\left(Q(F)\widehat{z}\right) - F\widehat{z} \end{aligned}$$

denote quantization errors. Note that the expressions for $\widehat{z}[r+1]$ and $\widehat{u}[r+1]$ were obtained by performing a quantization after each algebraic operation. A different software implementation can give rise to a different expression for the error. Although different software implementations lead to different quantization errors, e_{q1} and e_{q2} will always denote the difference between the quantized version in a particular implementation and the un-quantized version of the algebraic expressions.

The evolution, at the discrete-time instants $0, T, 2T, 3T, \ldots$, of the physical system (6) and (7) controlled by the implementation (12) can be obtained by combining the exact discretization of (6) and (7):

$$\begin{aligned} x[r+1] &= A_T x[r] + B_T u[r] & (13)\\ o[r+1] &= C x[r+1] & (14) \end{aligned}$$

with (12). The result is the discrete-time system:

$$\begin{aligned} x[r+1] &= A_T x[r] + B_T F\widehat{z}[r] + B_T e_{q2} & (15)\\ \widehat{z}[r+1] &= D_T\widehat{z}[r] + E_T C x[r] + e_{q1} & (16) \end{aligned}$$

Two new signals appear in (13) and (14). They are defined as follows:

$$x[r] = \xi(rT), \quad u[r] = \upsilon(rT), \quad \forall r \in \mathbb{N}_0.$$

3.4 Stability of perturbed systems

In this section we review a control theoretic result that will be used in our analysis. In order to state it, we first define stability formally.

DEFINITION 1. *A bounded set $S \subset \mathbb{R}^n$ is said to be globally asymptotically stable for a differential equation (5) if the following two properties hold:*

- $\forall \alpha \in \mathbb{R}^+ \ \exists t^* \in \mathbb{R}^+ \ \forall t \geq t^* \ d(\xi_x(t), S) \leq \alpha$;
- $\forall \alpha \in \mathbb{R}^+ \ \exists \beta \in \mathbb{R}^+_0 \ \forall t \in \mathbb{R}^+_0$ $d(x, S) \leq \beta \implies d(\xi_x(t), S) \leq \alpha$.

where the distance from x to S, denoted by $d(x, S)$, is defined by:

$$d(x, S) = \inf_{y \in S} \|x - y\|.$$

Whenever the set S contains a single point x_0, we speak of an asymptotically stable equilibrium point x_0 rather than of an asymptotically stable set. The first requirement in Definition 1 asks that for any initial state $x \in \mathbb{R}^n$, the corresponding trajectory ξ_x asymptotically converges to the set S in the sense that after $t^*(\alpha)$ units of time ξ_x will be α-close to S. The second condition prevents ξ_x from deviating too much from S when x is close to S. The notion of asymptotic stability for discrete-time difference equations is obtained from Definition 1 by replacing $t, t^* \in \mathbb{R}^+_0$ with $r, r^* \in \mathbb{N}_0$.

The following result describes how stability properties are affected by additive perturbations. It summarizes several results from [8] in a convenient form for the analysis in Section 3.5.

Proposition 1. *Consider the discrete-time linear system:*

$$x[r+1] = Ax[r]$$

and assume that the origin is an asymptotically stable equilibrium point. Then, for any signal $d : \mathbb{N} \to \mathbb{R}^n$ satisfying $\|d\| \le b(d)$ for some constant $b(d) \in \mathbb{R}_0^+$, the set:

$$S = \{x \in \mathbb{R}^n \mid \|x\| \le \gamma b(d)\}$$

is globally asymptotically stable for the discrete-time system:

$$x[r+1] \quad = \quad Ax[r] + Bd[r] \tag{17}$$

where γ is given by:

$$\gamma = \max_{\psi \in [0,2\pi[} \left\| (\mathsf{e}^{i\psi} 1_{n\times n} - A)^{-1} B \right\|.$$

with $i = \sqrt{-1}$. Moreover, the output $o = Cx$ is guaranteed to converge to the set:

$$S_C = \{o \in \mathbb{R}^p \mid \|o\| \le \gamma_C b(d)\} \tag{18}$$

with:

$$\gamma_C = \max_{\psi \in [0,2\pi[} \left\| C(\mathsf{e}^{i\psi} 1_{n\times n} - A)^{-1} B \right\|.$$

In the control literature, γ_C is known as the $\mathcal{L}_2$ gain of the control system. It describes how much the disturbance signal d is amplified by the discrete-time linear system. When a disturbance d of magnitude $b(d)$ is injected into the dynamical system, as described by (17), the state evolution will deviate from the nominal behavior by at most $\gamma_C b(d)$ as described by the set (18).

3.5 Stability analysis of controller implementations

The equations (15) and (16) describing the plant and the controller can be rewritten in matrix form:

$$w[r+1] = Gw[r] + He[r] \tag{19}$$

with:

$$w = \begin{bmatrix} x \\ \hat{z} \end{bmatrix}, \quad e = \begin{bmatrix} e_{q_1} \\ e_{q_2} \end{bmatrix},$$

and:

$$G = \begin{bmatrix} A_T & B_T F \\ E_T C & D_T \end{bmatrix}, \quad H = \begin{bmatrix} 0_{n\times n} & B_T \\ 1_{n\times n} & 0_{n\times n} \end{bmatrix}.$$

Under the working assumption that T is sufficiently small, it can be shown[1] that $w_0 = 0$ is an asymptotically stable equilibrium point for:

$$w[r+1] = Gw[r].$$

Using the state space representation in (19), it is straightforward to compute the set where the state trajectories asymptotically converge, as defined in the following proposition.

Proposition 2. *Consider the discrete-time linear system in (19). For any signals e_{q1} and e_{q2} satisfying $\|e_{q1}\| \le b(e_{q1})$ and $\|e_{q2}\| \le b(e_{q2})$, $b(e_{q1}), b(e_{q2}) \in \mathbb{R}_0^+$, the output $o = Mw$ is guaranteed to converge to the set:*

$$S_M = \{o \in \mathbb{R}^q \mid \|o\| \le \gamma_M (b(e_{q1}) + b(e_{q2}))\}$$

[1]Under a small sampling time T, matrix G can be seen as a small perturbation of the matrix e^{JT} with $J = \begin{bmatrix} A & BF \\ EC & D \end{bmatrix}$. Since the latter matrix has eigenvalues with negative real part, the eigenvalues of G are inside the unit disk.

with:

$$\gamma_M = \max_{\psi \in [0,2\pi[} \left\| M(\mathsf{e}^{i\psi} 1_{2n\times 2n} - G)^{-1} H \right\|$$

The constants $b(e_{q1})$ and $b(e_{q2})$ are bounds for $\|e_{q_1}\|$ and $\|e_{q_2}\|$, respectively. Note that the smaller the errors e_{q1} and e_{q2}, the smaller is the set where o converges to. In the limiting case where the quantization errors are non-existent, *i.e.*, $b(e_{q1}) = 0 = b(e_{q2})$, the set S becomes the point $o = 0$. Hence, if we seek an implementation guaranteeing that the set of outputs $o \in \mathbb{R}^q$ satisfying $\|o\| \le \rho$ is asymptotically stable, the quantization errors e_{q1} and e_{q2} in the software implementation need to satisfy:

$$b(e_{q1}) + b(e_{q2}) \le \frac{\rho}{\gamma_M}. \tag{20}$$

In the preceding equation, ρ describes how much the continuous variables are allowed to deviate from the nominal behavior due to implementation errors. For the example in Section 2 we have $T = 0.001$ and:

$$\begin{aligned} A_T &= \begin{bmatrix} 1 & 0.0010000 & -0.0010000 \\ -0.0000044 & 0.9999992 & 0.0000006 \\ 0.0000078 & 0.0000011 & 0.9999989 \end{bmatrix} \\ B_T &= \begin{bmatrix} 0.00000283 \\ 0.00566293 \\ 0.00000003 \end{bmatrix} \\ D_T &= \begin{bmatrix} 0.9999536 & 0.0790688 & -0.1272843 \\ -0.0010375 & 0.9893293 & -0.0011182 \\ 0.0000769 & 0.0014819 & 0.9887948 \end{bmatrix} \\ E_T &= \begin{bmatrix} -0.0781482 & 0.1262988 \\ 0.0086947 & 0.0014643 \\ -0.0014725 & 0.0111945 \end{bmatrix}. \end{aligned}$$

Since the objective is to regulate the distance between the locomotive and the car, described by x_1, the relationship between the output o and the state w is given by:

$$o = x_1 = [1_{1\times 1} \quad 0_{1\times 5}] \begin{bmatrix} x \\ z \end{bmatrix}$$

and the resulting value for γ_M is 36.11. Hence, for a desired value of $\rho = 2$cm, the quantization errors need to satisfy:

$$b(e_{q1}) + b(e_{q_2}) \le \frac{0.02}{36.11} = 5.53 \cdot 10^{-4}$$

In summary, our analysis has produced a quantitative relation given by Equation (20) between the implementation errors and the asymptotically stable set. In the following section, we show how bounds on the implementation errors $b(e_{q1})$ and $b(e_{q2})$ can be computed statically from the controller source code.

4. SYMBOLIC ERROR ANALYSIS

Intuition. Given a real-valued polynomial function $y = f(x)$, a program F implementing f using finite precision arithmetic, and a range $[l, u]$ for x, the goal of symbolic error analysis is to find how far the value $f(x)$ can be from the output of $F(\hat{x})$ when x is chosen from the range $[l, u]$ and $\hat{x}$ is the closest representation of x using the finite precision implementation of real numbers. To solve this problem, we first construct the verification condition $\mathsf{SP}(F)(\hat{x}, \hat{y})$ [23] for the function F which is a symbolic formula relating the input $\hat{x}$ to F with its output $\hat{y}$. Then, we set up a set of constraints that is the conjunction of: (a) "x is chosen in its range" $x \in [l, u]$, (b) "x differs from $\hat{x}$ in at most the precision δ of the finite precision representation" $|x - \hat{x}| \le \delta$, (c) $y = f(x)$, (d) "the program F transforms $\hat{x}$ to $\hat{y}$" $\mathsf{SP}(F)(\hat{x}, \hat{y})$, and finally, ask (e) what is the maximum difference between y and $\hat{y}$ under the constraints (a)-(d)?

The rest of this section formalizes this intuition.

s	$\mathsf{SP}_{\mathbb{R}}(s,\theta)$
$x := c$	$\exists x'.\theta[x'/x] \wedge x = c$
$x := y \oplus z$	$\exists x'.\theta[x'/x] \wedge x = y \oplus z$
$\mathtt{assume}(e)$	$\theta \wedge e$
$s_1; s_2$	$\mathsf{SP}_{\mathbb{R}}(s_2, \mathsf{SP}_{\mathbb{R}}(s_1,\theta))$
$s_1 \talloblong s_2$	$\mathsf{SP}_{\mathbb{R}}(s_1,\theta) \vee \mathsf{SP}_{\mathbb{R}}(s_2,\theta)$

Table 1: Strongest postconditions for real-valued programs

4.1 Real-Valued Programs

We represent (mathematical) control functions as *real-valued programs.* A real-valued program $P = (I, L, O, \mathsf{rng}, s)$ consists of a set of *input variables* I, a set of *local variables* L, a set of *output variables* O, a function rng mapping each input variable i in I to an interval $[l_i, h_i]$ of the reals, and a body s defined by the grammar:

$$s \ ::= \ x := c \mid x := y \oplus z \mid s_1; s_2 \mid \mathtt{assume}(e) \mid s_1 \talloblong s_2 \qquad (21)$$

where x ranges over variables in $L \cup O$, y, z range over variables in $I \cup L$, c ranges over arbitrary rational constants, e ranges over Boolean expressions over the variables in $I \cup L$, and $\oplus \in \{+, -, *\}$ ranges over arithmetic operations. The body s consists of assignment statements, sequential composition $s_1; s_2$, conditionals ($\mathtt{assume}$), and non-deterministic choice $s_1 \talloblong s_2$. For x, z variables and c a rational constant, we write $x := c * z$ as shorthand for $y := c; x := y * z$.

Note that there are no loops in the language. This is because most programs in this domain come with *static* bounds on the number of iterations of a loop, and so loops can be unrolled statically. We keep the non-deterministic choice operation to model external conditions that choose between different implementations. For simplicity of exposition, we omit arrays and pointers as well as non-recursive function calls from our language, although our implementation handles these features.

For a set X of variables, an X-valuation is a mapping from X to the reals. We write $\{\{X\}\}_{\mathbb{R}}$ for the set of all X-valuations. For an X-valuation ν and variable $x \in X$, we write $\nu(x)$ for the value of x under the valuation ν. A *program state* is a $I \cup L \cup O$-valuation. In the following, we use first-order formulas with free variables in $I \cup L \cup O$ to denote sets of program states: a formula θ represents the set of program states that satisfy the formula θ.

The semantics of a real-valued program is given using the strongest post-condition operation $\mathsf{SP}_{\mathbb{R}}$. The function $\mathsf{SP}_{\mathbb{R}}$ maps a body s and a first-order formula θ to a first-order formula $\mathsf{SP}_{\mathbb{R}}(s,\theta)$. We use the notation $\theta[x'/x]$ to denote the formula θ in which each occurrence of x is substituted by x'. For a first-order formula θ with free variables in X, and an X-valuation ν, we say ν *satisfies* θ if the formula obtained by substituting each occurrence of $x \in X$ with $\nu(x)$ evaluates to true. In the following, we identify an X-valuation ν with a $|X|$-dimensional vector. For a first-order formula θ and a set $X = \{x_1, \ldots, x_k\}$ of variables, we write $\exists X.\theta$ as shorthand for $\exists x_1.\exists x_2 \ldots \exists x_k.\theta$.

The transformer $\mathsf{SP}_{\mathbb{R}}$ is shown in Table 1. The operations have the usual semantics [23], and $\mathsf{SP}_{\mathbb{R}}(s,\theta)$ returns a formula that characterizes the set of program states that can be reached by executing the statement s from a program state satisfying the formula θ. An I-valuation $\nu \in \{\{I\}\}_{\mathbb{R}}$ satisfies the rng constraints if it maps every $v \in I$ to a value in $\mathsf{rng}(v)$, i.e., if $\nu(v) \in \mathsf{rng}(v)$ for each $v \in I$. The semantics of a real-valued program defines a mathematical relation, written $[\![P]\!]_{\mathbb{R}}$, between I-valuations and O-valuations in the following way: the pair of valuations $(\nu, \mu) \in [\![P]\!]_{\mathbb{R}}$ is in the relation if ν is an I-valuation, μ is an O-valuation, and μ satisfies $\exists L.\mathsf{SP}_{\mathbb{R}}(s, \bigwedge_{v \in I} v = \nu(v))$.

4.2 Fixed-Point Representation

A *fixed-point type* is a triple $\langle s, n, m \rangle$ consisting of a *sign bit* $s \in \{\mathtt{s}, \mathtt{u}\}$ (for *signed* and *unsigned*), a *length* $n \in \mathbb{N}$, and a *fractional part* $m \in \mathbb{N}$. A fixed-point type $\langle s, n, m \rangle$ interprets a bitvector of n bits as a rational number in the following way. A bitvector $b = b_{n-1} \ldots b_0$ of type $\langle \mathtt{u}, n, m \rangle$ represents the rational number

$$\frac{1}{2^m} \sum_{i=0}^{n-1} 2^i b_i$$

i.e., the bitvector represents a rational number where the last m bits are used for the fractional part, and the top $n - 1 - m$ bits for the integer part. The *range* of possible values for the type $\langle \mathtt{u}, n, m \rangle$ is the set $U_{n,m} = \{\frac{p}{2^m} \mid p \in \mathbb{N}, 0 \le p \le 2^n - 1\}$.

The bitvector $b_{n-1} \ldots b_0$ of type $\langle \mathtt{s}, n, m \rangle$ represents the rational number

$$\frac{1}{2^m} \left[-2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i \right]$$

Thus, a signed fixed-point number represents a positive, zero, or negative rational number. The *range* of possible values for the type $\langle \mathtt{s}, n.m \rangle$ is the set $S_{n,m} = \{\frac{p}{2^m} \mid p \in \mathbb{Z}, -2^{n-1} \le p \le 2^{n-1} - 1\}$. The most significant bit b_{n-1} of a fixed-point number of type $\langle \mathtt{s}, n, m \rangle$ is called the *sign* bit.

Representation of constants. Since not all real numbers can be represented using fixed-point types, we represent fixed-point *approximations* to real numbers. Let $\langle \mathtt{u}, n, m \rangle$ be a fixed-point type, and let $0 \le c < 2^{n-m}$ be a real number. The *representation* $\mathsf{repr}(c) : \langle \mathtt{u}, n, m \rangle$ of c is the number $\frac{p}{2^m}$ for $0 \le p \le 2^n - 1$ for which $c - \frac{p}{2^m}$ is minimized. Clearly, $(c - r) \le \frac{1}{2^m}$ for the representation r of c. Similarly, for the fixed-point type $\langle \mathtt{s}, n, m \rangle$ and a rational $-2^{n-1-m} \le c < 2^{n-m-1}$, the *representation* $\mathsf{repr}(c) : \langle \mathtt{s}, n, m \rangle$ of c is the number $\frac{p}{2^m}$ for $-2^{n-1} \le p \le 2^{n-1} - 1$ for which $c - \frac{p}{2^m}$ is minimized. Again, $(c - r) \le \frac{1}{2^m}$ for the representation r of c.

Let X be a set of variables and let typ be a function mapping each $x \in X$ to a fixed-point type. For a valuation $\nu \in \{\{X\}\}_{\mathbb{R}}$ mapping each variable in X to a real value, we write $\mathsf{repr}(\nu)$ for the fixed-point valuation that maps each variable x to $\mathsf{repr}(\nu(x))$ of type $\mathsf{typ}(x)$.

Arithmetic operations and assignments. Consider the typed variables $x : \langle \mathtt{u}, n_x, m_x \rangle$, $y : \langle \mathtt{u}, n_y, m_y \rangle$, and $z : \langle \mathtt{u}, n_z, m_z \rangle$, and the assignment $x := y + z$. In fixed-point arithmetic, the addition must be performed by first scaling the bitvectors y and z so that their binary points align. Moreover, the addition can result in a number with $\max\{n_y, n_z\} + 1$ bits, and this number must be scaled to fit into the bitvector x. Table 2 shows the sequence of bitvector operations to perform these steps.

We consider the case $m_y \le m_z$ and $n_y \le n_z$, the other cases are similar. First, we multiply y by $2^{m_z - m_y}$ to align the binary points of the two operands (variable t_1 in Table 2). Second, the variables t_1 and z are added as bitvectors. This creates a number t_2 with m_z fractional bits, and $n_z - m_z + 1$ integer bits. These bits must be "fitted" into the n_x bits of x. Assuming $m_x \le m_z$ (the other cases are similar), we truncate the m_z fractional bits by right-shifting the bitvector $m_z - m_x$ bits and storing the result in t_3 (thus keeping m_x bits of precision for the fractional part). Finally, we copy the lower n_x bits of t_3 into the bitvector x. The subtraction operation has to consider two cases. If x is greater than z, in which case the operation proceeds similar to addition. If x is less than z, then by 2's complement arithmetic, a 2^n term is added to the result.

Command	Semantics	Assumptions
$x : \langle \mathtt{u}, n_x, m_x \rangle := c$ $x : \langle \mathtt{s}, n_x, m_x \rangle := c$	$x := \mathsf{repr}(c) : \langle \mathtt{u}, n_x, m_x \rangle$ $x := \mathsf{repr}(c) : \langle \mathtt{s}, n_x, m_x \rangle$	$0 \leq c < 2^{n_x - m_x}$ $-2^{n_x - m_x - 1} \leq c < 2^{n_x - m_x - 1}$
$x : \langle \mathtt{u}, n_x, m_x \rangle :=$ $y : \langle \mathtt{u}, n_y, m_y \rangle \pm z : \langle \mathtt{u}, n_z, m_z \rangle$	$t_1 : \langle \mathtt{u}, n_z + 1, m_z \rangle := \mathsf{shl}(y, m_z - m_y);$ $t_2 : \langle \mathtt{u}, n_z + 1, m_z \rangle := t_1 \pm z;$ $t_3 : \langle \mathtt{u}, n_z + 1, m_x \rangle := \mathsf{shr}(t_2, m_z - m_x);$ $x : \langle \mathtt{u}, n_x, m_x \rangle := \mathsf{lsb}(t_3, n_x);$	$m_y \leq m_z,\ n_y \leq n_z\ m_x \leq m_z$
$x : \langle \mathtt{u}, n_x, m_x \rangle :=$ $y : \langle \mathtt{u}, n_y, m_y \rangle * z : \langle \mathtt{u}, n_z, m_z \rangle;$	$t_1 : \langle \mathtt{u}, n_y + n_z, m_y + m_z \rangle = y * z;$ $t_2 : \langle \mathtt{u}, n_y + n_z, m_x \rangle = \mathsf{shr}(t_1, m_y + m_z - m_x);$ $x : \langle \mathtt{u}, n_x, m_x \rangle = \mathsf{lsb}(t_2, n_x)$	$m_x \leq m_y + m_z$

Table 2: Semantics of unsigned fixed-point operations in terms of bitvector operations. The other cases are symmetric. The shr and shl operators are bitvector shift-right and shift-left operations, and $\mathsf{lsb}(x, k)$ picks the lower order k bits of a bitvector x.

While we can give a direct semantics for signed arithmetic operations using signed bitvector operations, it is conceptually easier to give the semantics for signed numbers by reduction to unsigned ones. We do this by separately tracking the sign and the magnitude of a signed number, performing the operations on the magnitudes using unsigned arithmetic, and finally putting the appropriate sign bits back. We omit the details for lack of space.

To perform fixed-point multiplication of y and z, we multiply y and z as signed or unsigned integers to get a bitvector with $n_y + n_z$ bits in the unsigned case (and $n_y + n_z + 1$ bits in the signed case) of which $m_y + m_z$ bits represent the fractional part. These bits are "fitted" into x by first truncating the fractional part to keep m_x bits of precision, and then copying the lower n_x bits into x (and copying the sign bit into the sign bit of x). The corresponding bitvector operations are shown in Table 2.

4.3 Fixed-point Semantics

A fixed-point program $P = (I, L, O, \mathsf{typ}, s)$ consists of sets I, L, O of *input*, *local*, and *output* variables respectively, a type-map typ from $I \cup L \cup O$ to fixed-point types, and a body s defined by the grammar (21). That is, a fixed-point program is syntactically identical to a real-valued program, but each variable is interpreted as a fixed-point type rather than a rational number. We assume that programs are well-typed in that arithmetic operations do not mix signed and unsigned types.

For a set X of fixed-point variables, an X-fixed-point valuation is a function that maps each variable $v \in X$ with $\mathsf{typ}(v) = \langle \mathtt{u}, n, m \rangle$ (respectively, $\mathsf{typ}(v) = \langle \mathtt{s}, n, m \rangle$) to a value in $U_{n,m}$ (respectively, $S_{n,m}$). We write $\{\{X\}\}$ for the set of all X-fixed-point valuations. When the type of variables (rational or fixed-point) is clear from the context, we omit "fixed-point" and simply write valuation. A fixed-point program state is an $I \cup L \cup O$-fixed-point valuation. Notice that fixed-point programs are finite-state, since the possible set of fixed-point valuations to all the fixed-point variables in a program is finite.

The semantics of a fixed-point program is given using the strongest post-condition operation SP mapping a body s and a first-order formula θ to a first-order formula $\mathsf{SP}(s, \theta)$. The definition of SP is identical to $\mathsf{SP}_{\mathbb{R}}$ for the constructs ;, $[\!]$, and $\mathtt{assume}(e)$. The strongest postcondition operation for assignments is defined using the semantics of arithmetic operations and assignments given in Table 2.

A fixed-point program P defines a relation $[\![P]\!]$ between I-valuations and O-valuations: the pair $(\nu, \mu) \in [\![P]\!]$ if $\nu \in \{\{I\}\}$ is an I-valuation, $\mu \in \{\{O\}\}$ is an O-valuation, and μ satisfies $\exists w \in L.\mathsf{SP}(s, \bigwedge_{v \in I} v = \nu(v))$.

4.4 Symbolic Error Analysis

We now formally define the error between a real-valued program and its fixed-point implementation.

Definition 2. *The* implementation-error decision problem *asks, given a real-valued program $P = (I, L, O, \mathsf{rng}, s)$, a fixed-point program $P' = (I, L', O, \mathsf{typ}, s')$, and an error bound $\epsilon > 0$, is it true that for every $\mu \in \{\{I\}\}_{\mathbb{R}}$ satisfying the* rng *constraints, we have $\|\nu - \nu'\| < \epsilon$ for every pair $\nu \in \{\{O\}\}_{\mathbb{R}}$, $\nu' \in \{\{O\}\}$ satisfying $[\![P]\!]_{\mathbb{R}}(\mu, \nu)$ and $[\![P']\!](\mathsf{repr}(\mu), \nu')$? (Note that P and P' have the same set of input and output variables, but can have different local variables and different bodies.)*

Intuitively, for an input valuation μ satisfying the rng constraints, we run P on μ and P' on $\mathsf{repr}(\mu)$ and compare the results. The answer to the implementation-error decision problem is "yes" if the norm of the difference of the outputs between the programs is less than ϵ.

We now show that the implementation-error decision problem reduces to the satisfiability problem for the combination theory of bitvectors and reals with addition and multiplication. For a variable $v \in I \cup O$, we write v as the real-valued variable in P and $\mathsf{fp}(v)$ for the fixed-point version of v in P'. For a formula φ and a set X of variables, we write $\varphi[\mathsf{fp}(X)/X]$ for the formula obtained by replacing each variable $x \in X$ appearing φ with $\mathsf{fp}(x)$. Consider the following formula:

$$\begin{array}{ll}
\bigwedge_{v \in I} v \in \mathsf{rng}(v) \wedge & \text{(a)} \\
\bigwedge_{v \in I : \mathsf{typ}(v) = \langle \cdot, n, m \rangle} |v - \mathsf{fp}(v)| \leq \frac{1}{2^m} \wedge & \text{(b)} \\
\mathsf{SP}_{\mathbb{R}}(s, \mathit{true}) \wedge & \text{(c)} \\
\mathsf{SP}(s', \mathit{true})[\mathsf{fp}(I \cup L' \cup O)/(I \cup L' \cup O)] & \text{(d)} \\
\Rightarrow & \\
\quad \|\mathsf{o} - \mathsf{fp}(\mathsf{o})\| \leq \epsilon & \text{(e)}
\end{array} \tag{22}$$

where o denotes the vector of all elements of O. In the formula, (a) encodes the rng constraints on the variables in I, (b) encodes the quantization error in the inputs, (c) and (d) encode the semantics of the programs P and P' respectively, and (e) encodes that the output differences are bounded by ϵ. The answer to the implementation-error decision problem is "yes" iff the above formula is valid.

By naive enumeration, the bitvector constraints in the above formulas can be reduced to Boolean operations. Thus, the implementation-error decision problem reduces to a satisfiability question in the theory of reals with multiplication (naively, by enumerating all of the finitely many bits). The decidability of this latter theory [22, 7] implies the following theorem.

Theorem 1. *The implementation-error decision problem is PSPACE-complete.*

Note that in order to prove the error bound is *not* ϵ, we can (1) guess the bits in polynomial time, and (2) solve the resulting problem in the existential theory of reals with addition and multiplication [7]. This gives a PSPACE bound for the problems, using Savitch's theorem and closure of PSPACE under complementation. The PSPACE lower bound is from the PSPACE lower bound for the existential theory of reals.

In practice, instead of enumeration, in our implementation, we use efficient decision procedures for the combination theory.

While we assume that ϵ is given as an input, notice that we can approximate the minimal ϵ that bounds the error to any desired precision by using binary search in a range.

Linear Approximation. The solution to the absolute-error problem uses a reduction to a decision procedure for *non-linear* arithmetic. In practice, these decision procedures are not as scalable as decision procedures for *linear* arithmetic. We now consider a special case of the problem for which we can reduce the error problems to problems in linear arithmetic.

A *linear* constraint is of the form $\sum_i c_i x_i \sim b$ where c_i and b are real numbers and $\sim \in \{\geq, \leq, =\}$. A linear formula is a Boolean combination of linear constraints. As *linear* real-valued program $P = (I, L, O, \mathsf{rng}, s)$, every assignment statement in s is one of the forms $x := c$, $x := y \pm z$, or $x := c * y$ for variables x, y, z and real constant c, and in every assume statement $\mathtt{assume}(e)$, we have e is a linear constraint.

Linear real-valued programs are an important special case: the implementation of a linear controller or a non-linear controller through lookup tables and linear interpolation is a linear real-valued program.

For linear real-valued programs, the constraints $\mathsf{SP}_{\mathbb{R}}(s, \mathit{true})$ is a formula in linear arithmetic. Thus, in the constraints (22), the antecedent is a formula in the combination theory of linear arithmetic and bitvectors. The problem is that we use the Euclidean norm, which introduces non-linear terms.

We define the *linear approximation* to the implementation-error problem in which we ask if the 1-norm of the output difference in less than ϵ, i.e., if $\|\nu - \nu'\|_1 < \epsilon$ in Definition 2 instead of the Euclidean norm. For linear real-valued programs, the linear approximation to the problem can be reduced to a decision problem in the combination theory of linear arithmetic and bitvectors. This gives the following result.

THEOREM 2. *For a linear real-valued program P and a fixed-point program P', the linear approximation to the implementation-error problem is coNP-complete.*

To show that an error bound is greater than ϵ, we can guess the bits and reduce the problem to the satisfiability question for linear arithmetic. Thus, the complement of the problem is in NP. coNP-completeness follows from the hardness of Boolean reasoning.

Notice that the linear approximation can be used to provide a conservative upper bound on the Euclidean norm, using the fact that $\|x\| \leq \|x\|_1$ for any vector x.

4.5 Arithmetic Encoding

While the bitvector semantics provides an "execution semantics" for fixed-point programs, we now give an alternate semantics to fixed-point programs by reduction to constraints over integers that is more suited to symbolic analysis. In the arithmetic semantics, we associate an integer variable $\hat{x}$ with each fixed-point variable x. The key idea is to simulate the bitvector operations on integers.

The fixed-point implementation of an arithmetic operation in $\{+, -, *\}$ involves the arithmetic operation, and optionally one or more shift operations and an lsb operation in the fixed-point code. In the integer encoding, after each core arithmetic operation, we separate the sign and magnitude of the result before applying shift and lsb operations on it. Let t be the temporary variable representing the result of the arithmetic operation. We represent by $\mathsf{sgn(t)}$ the sign of t and by $\mathsf{mgn(t)}$ the magnitude of t. The magnitude $\mathsf{mgn(t)}$ can be considered an unsigned fixed-point number with the same length and the same number of fraction bits as t. Thus, we only need to simulate the bitvector operations on unsigned bitvectors using arithmetic operations (and using the Boolean structure to encode the different possibilities on the sign bits).

Now let x be an unsigned number. We simulate $\mathsf{shl}(x, \ell)$ by $\hat{x} * 2^\ell$ (note that ℓ is a constant, so this is multiplication by a constant). We simulate $\mathsf{shr}(x, \ell)$ by $(\hat{x} - (\hat{x} \mod 2^\ell))/2^\ell$. We simulate $\mathsf{lsb}(x, \ell)$ by $\hat{x} \mod 2^\ell$. Note that the shift operations and the lsb operation does not modify the sign of the operand. So now we can determine the sign of the result of the whole arithmetic operation to be $\mathsf{sgn(t)}$. While $x \mod k$ is not directly implemented as an operation in a linear arithmetic decision procedure, the predicate $y = x \mod k$ is easily encoded as $\exists z. x = kz + y \wedge 0 \leq y < k$.

5. EXTENSIONS

We now indicate two extensions to the analysis in Section 3.

Sensor and Actuator Errors. The first extension deals with errors arising from sensors and actuators in the system.

Sensing errors e_{sense} can be defined as the mismatch $e_{sense} = \widehat{o}[n] - o[n]$ between the measured output $\widehat{o}[n]$ of the sensor and the true value $o[n]$ of the signal. Similarly, actuation errors are defined by the mismatch $e_{act} = \widehat{u}[n] - u[n]$ between the desired actuator value $u[n]$ and the actual value $\widehat{u}[n]$ enforced by the actuator. Bounds on these errors are readily available from the sensor and actuator specifications. Using e_{sense}, we can redefine the quantization error e_{q1} as:

$$e_{q1} = Q(Q(D_T)\widehat{z} + Q(E_T)Q(o + e_{sense})) - D_T\widehat{z} + E_T o.$$

Similarly, we can redefine the quantization error e_{q2} as:

$$e_{q2} = Q(Q(F)\widehat{z}) + e_{act} - F\widehat{z}.$$

With these new definitions for e_{q1} and e_{q2}, the analysis in Section 3.5 remains valid and the implemented controller is guaranteed to steer the state of the plant to the set of states x satisfying (20).

Nonlinear Systems. The analysis of Section 3 can be extended to nonlinear control systems by performing an analysis based on *Lyapunov functions*. A Lyapunov function $V : \mathbb{R}^n \to \mathbb{R}_0^+$ satisfies $V(x) = 0 \implies x = 0$ and $\frac{\partial V}{\partial x} f(x, k(x)) \leq -\lambda V(x)$. It is known that the existence of a Lyapunov function implies that $x_0 = 0$ is an asymptotically stable equilibrium point.

We now illustrate how our analysis for implementation errors can be done for feedback controllers of the form $\upsilon = k(\xi)$ designed for a nonlinear control system $\frac{d}{dt}\xi = f(\xi, \upsilon)$. When designing controller k with the objective of rendering $x_0 = 0$ a globally asymptotically stable equilibrium point, a Lyapunov function for $\frac{d}{dt}\xi = f(\xi, k(\xi))$ is also designed. Moreover, if the controller is *robust*, a typical requirement for controller design, the following stronger inequality also holds:

$$\frac{\partial V}{\partial x} f(x, k(x) + e) \leq -\lambda V(x) + \sigma \|e\|^2.$$

Due to the implementation errors, the actuators receive the value:

$$u = Q(k(Q(x))) = k(x) + \varepsilon_{q3}$$

where the quantization error ε_{q3} is defined by:

$$\varepsilon_{q3} = Q(k(Q(x))) - k(x).$$

Hence, the time derivative of $V \circ \xi$ will be given by:

$$\frac{d}{dt} V \circ \xi = \frac{\partial V}{\partial x}\Big|_{x=\xi} f(\xi, k(\xi) + \varepsilon_{q3}) \leq -\lambda V \circ \xi + \sigma \|\varepsilon_{q3}\|.$$

Through integration we arrive at:

$$\begin{aligned} V \circ \xi(t) &\leq e^{-\lambda t} V \circ \xi(0) + \int_0^t e^{-\lambda(t-\tau)} \sigma \|\varepsilon_{q3}(\tau)\| d\tau \\ &\leq e^{-\lambda t} V \circ \xi(0) + \frac{\sigma b(\varepsilon_{q3})}{\lambda} \end{aligned}$$

where $b(\varepsilon_{q3})$ is an upper bound for $\|\varepsilon_{q3}\|$. Since:

$$\lim_{t\to\infty} V \circ \xi(t) \leq \sigma \frac{b(\varepsilon_{q3})}{\lambda}$$

the trajectories of the controlled system are guaranteed to converge to the set of states x defined by $V(x) \leq \sigma b(\varepsilon_{q3})/\lambda$. As expected, the implemented controller still enforces asymptotic stability. However, in the presence of implementation errors, we can only guarantee that trajectories converge to a set containing $x_0 = 0$ and the size of this set decreases when the error ε_{q3} is reduced.

6. EVALUATION

6.1 Implementation

We have implemented Costan, an automatic tool to compute the error bound between a mathematical control law and a fixed-point implementation for the control law. We model the mathematical control system in Simulink, and generate fixed-point controller code using Simulink's Fixed-Point Advisor and Real-Time Workshop. The real-valued program for the controller is extracted from the Simulink model of the controller as an imperative program.

The inputs to Costan consist of the mathematical model and the fixed-point implementation of the controller, a mapping between the input and output variable names used in the two descriptions, the ranges of values for the input variables, and the fixed-point types for each variable in the fixed-point implementation. Currently, the ranges for the input variables are determined by Matlab simulations. Ranges for the variables measured and computed by the controller can also be obtained by analyzing the mathematical models. Given a desired operating region for the physical system, the control designer can compute how much the physical variables will deviate from this region while converging to the desired equilibrium point. The possibility of computing such value is essentially a consequence of the second requirement in Definition 1. A similar analysis can then be performed for the observer in order to determine the range for all the variables in the control code. The fixed-point types can be obtained from Simulink Fixed-point advisor, or given as inputs to the Fixed-point code generator.

The core of Costan generates verification conditions as well as the constraints for the implementation-error decision problem shown in Equation (22). Loops are statically unrolled. In most cases, there is an explicit constant bound on the number of iterations. A few loops, e.g., those implementing binary search over a lookup-table to look up precomputed values, do not have an explicit constant bound in the code, but these bounds can be inserted from the (static) knowledge of the size of the lookup table.

Our tool uses the decision procedures Yices [11] and HySat [16]. We use Yices to solve the linear approximation to the implementation-error problem for linear controllers. For nonlinear controllers, we use HySat. For a given ϵ, if the constraints from Equation (22) are not valid, we get a concrete test input that indicates where the mathematical controller and the implementation diverges. We use the integer-based encoding for fixed-point arithmetic over the bitvector implementation. This is because in our experiments (described below), the bitvector encoding scaled poorly.

Starting with a range for the error bound, Costan performs a binary search over the range to find out the smallest ϵ for which the difference between the outputs is less than ϵ. As a terminating condition we specify a margin on the range. If at any iteration the size of the range for ϵ goes below this margin then we terminate the algorithm deciding the upper bound of current range to be the error bound.

Example	Error bound	Set size (ρ)	Run time
vehicle steering (16bit)	0.0163	0.0375	1m14.313s
pendulum (16bit)	0.0508	0.1806	2m36.409s
dc motor (16bit)	0.0473	1.0889	2m15.110s
train car - 1 car (32bit)	5e-7	2.6080e-5	3m25.478s
train car - 2 cars (32bit)	1.5e-6	9.4000e-5	5m39.607s
train car - 3 cars (32bit)	8.5e-6	0.0010	9m34.485s
train car - 4 cars (32bit)	3.351e-5	0.0080	10m9.179s
train car - 5 cars (32bit)	1.655e-4	0.0627	20m28.822s
jet engine[poly] (16bit)	4e-3	0.0230	0m0.551s
jet engine[3×8]	6.40	37.0431	0m34.636s
jet engine[5×10]	4.48	25.9296	0m34.293s
jet engine[7×14]	2.73	15.8009	1m6.981s
jet engine[21×21]	1.25	7.2348	18m15.794s
jet engine[21×101]	0.88	5.0933	50m23.127s
jet engine[100×100]	0.33	1.9100	103m19.977s

Table 3: Experimental Results

In most examples, the mathematical controller and the fixed-point implementation had identical syntactic structure. This allowed us to perform compositional analysis, in which we computed the best error bounds on all live variables going out of a block of code based on computed error bounds on all variables reaching the block of code. However, our analysis cannot be fully compositional, as the mathematical controller and the implementation can take different branches at a conditional. So, at conditionals, our analysis "fell back" to verification condition generation.

6.2 Experiments

We have applied Costan to a set of linear and nonlinear controller implementations to estimate the implementation errors and hence, using the analysis of Section 3, the region of asymptotic stability for the systems. Table 3 shows our experimental results on linear and nonlinear controllers, including the set $S = \{||x|| \leq \rho\}$ to where the state trajectories converge.

Linear Controllers. As the first example we choose the vehicle steering model from [3]. This model describes how to control the trajectory of a vehicle through an actuator that causes a change in the orientation. It is possible to design a linear feedback controller that stabilizes the dynamics and tracks a given reference value r of the lateral position of the vehicle. The inputs to the controller are the reference r and two state inputs coming from the controlled system. The output of the controller is the control signal that goes as an input to the vehicle steering system. For the reference input we choose the range to be $[0, 100]$. We considered a 16-bit implementation. Our analysis finds that the absolute error for this fixed-point C code is bounded above by 0.0163.

We also demonstrate the scalability of the integer encoding over the direct bitvector encoding. Figure 2 shows how the time due to decision procedure varies with ϵ for bitvector-based and the integer-based encodings. The SAT-solving time grows rapidly near the "best" ϵ, and the bitvector analysis timed out in the range $[0.012, 0.8]$ while the integer encoding was stable.

We considered two other "textbook" linear control systems with feedback: the simple inverted pendulum [19] and the armature-controlled DC motor system [24]. For the simple inverted pendulum, the controller has three inputs: one is the reference input and the other two are the state inputs. Our analysis reveals that the absolute error bound for a 16-bit implementation of the controller is 0.0508. The controller for the DC motor system has four inputs: the reference input and three state inputs. The error bound obtained for the DC motor controller output for a 16-bit implementation is 0.0473.

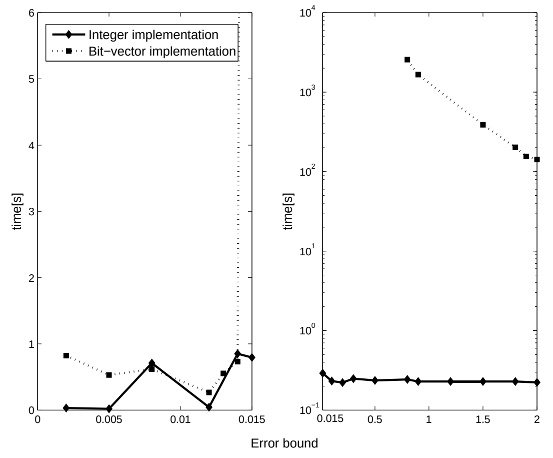

Figure 2: Decision procedure runtime for bitvector and integer implementation

In the three examples described above the states of the system were directly measured. To consider a system where only some of the states can be measured, we revisit the example in Section 2. We scale up the example gradually by adding up to 5 cars. The measured states correspond to the velocities of the locomotive and the train cars. The states that cannot be measured correspond to the distance between the locomotive and the first car and the distances between any two cars. Hence, for n train cars we have $n+1$ measured states and n states that are estimated using the measured states. The inputs to the controller are the measured states, while the control output, the force applied to the engine, is computed from the measured inputs and the estimated states. We consider 32-bit fixed-point representation for all the variables. The length of the fraction part of each variable is decided in such a way that no overflow occurs and the precision of the variable is maximized. Table 3 shows the error bounds for train-car controllers with up to 5 cars. The model with 5 cars (with 11 states) can be analyzed in about 20 minutes. The 32-bit implementation of the controller helps us achieve significantly small error bound.

While generating fixed-point code for a controller, a few times we have erroneously introduced overflow in some variables. Costan has been able to detect the overflow by detecting that the desired error bound is impossible to achieve. From the counterexample obtained from the decision procedure we were able to detect the overflow in the implementation.

Nonlinear Controllers. We analyzed two implementations of a jet-engine controller from [20]. There are two inputs, and the control law is a polynomial function of the inputs. One implementation directly evaluates the polynomial control law using fixed-point arithmetic. The other pre-computes the control law for various values of the input, and looks up the control action for an input from the "closest" point in the table. In both the cases we consider 16-bit implementation. When the controller is implemented as a lookup table, apart from the quantization error, error is also introduced due to the approximation of the output values in the lookup table. The error bound on the output of the controller based on lookup tables is dominated by the error in the lookup table, which depends on the size of the table. We used Costan to compute the error bounds on the control output for lookup tables of different size. Table 3 shows error bound obtained for different dimensions of the lookup table (e.g., the example "jet engine[3×8]" denotes a controller with a lookup table with $3 \times 8 = 24$ entries). For larger lookup tables, our implementation adopts a compositional strategy of dividing up the input range, computing output errors in each range, and then taking the maximum of the results. Calculation of the error bound on the single lookup table takes more than 2 hours for the lookup table with 2121 entries, while we can get the same error bound in less than one hour by dividing the lookup table into four smaller tables. The largest lookup table that we analyze has 10000 entries, we break it into 16 lookup table of equal size and successfully calculate the error bound in less than 2 hours. As might be expected, the direct polynomial evaluation with 16-bit arithmetic is more precise than even our largest lookup table implementation.

7. REFERENCES

[1] F. Alegre, E. Feron, and S. Pande. Using ellipsoidal domains to analyze control systems software. *CoRR*, abs/0909.1977, 2009.

[2] P.J. Antsaklis and A.N. Michel. *Linear Systems*. McGraw-Hill, 1997.

[3] K. J. Astrom and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Princeton and Oxford, 2009.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.

[5] O. Bouissou, E. Goubault, S. Putot, K. Tekkal, and F. Védrine. HybridFluctuat: A static analyzer of numerical programs within a continuous environment. In *CAV*, LNCS 5643, pages 620–626. Springer, 2009.

[6] A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *FMCAD*, pages 69–76. IEEE, 2009.

[7] J. Canny. Some algebraic and geometric computations in PSPACE. In *STOC*, pages 460–467. ACM, 1988.

[8] T. Chen and B.A. Francis. *Optimal Sampled-Data Control Systems*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1995.

[9] P. Cousot. Integrating physical systems in the static analysis of embedded control software. In *APLAS*, pages 135–138, 2005.

[10] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, LNCS 5825, pages 53–69. Springer, 2009.

[11] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, LNCS 4144, pages 81–94. Springer, 2006.

[12] G. Fainekos, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Robustness of model based simulation. In *Real Time Systems Symposium*, pages 345–354. IEEE, 2009.

[13] J. Feret. Static analysis of digital filters. In *ESOP*, LNCS 2986, pages 33–48. Springer, 2004.

[14] E. Feron and F. Alegre. Control software analysis, part I open-loop properties. *CoRR*, abs/0809.4812, 2008.

[15] E. Feron and F. Alegre. Control software analysis, part II closed-loop analysis. *CoRR*, abs/0812.1986, 2008.

[16] M. Fränzle, C. Herde, S. Ratschan, T. Schubert, and T. Teige. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *J. SAT*, 1:209–236, 2007.

[17] E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In *FMICS*, LNCS 4916, pages 3–20. Springer, 2007.

[18] E. Goubault, S. Putot, and M. Martel. Some future challenges in the validation of control systems. In *ERTS*, 2006.

[19] B. Kisacanin and G. C. Agarwal. *Linear Control Systems*. Kluwer Academic/Plenum Publishers, 2002.

[20] M. Krstic and P.V. Kokotovic. Lean backstepping design for a jet engine compressor model. *4th IEEE Conference on Control Applications*, pages 1047–1052, 1995.

[21] P. McLane, L. Peppard, and K. Sundareswaran. Decentralized feedback controls for the brakeless operation of multilocomotive powered trains. *IEEE Transactions on Automatic Control*, 21(3):358–363, 1976.

[22] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley and Los Angeles, 1951.

[23] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, 1993.

[24] S. H. Zak. *Systems and Control*. Oxford University Press, New York and Oxford, 2003.