

# Systematic Testing for Control Applications

Rupak Majumdar  
UC Los Angeles and MPI-SWS  
rupak@cs.ucla.edu

Indranil Saha  
UC Los Angeles  
indranil@cs.ucla.edu

Zilong Wang  
Nankai University, China  
wangzilong@mail.nankai.edu.cn

**Abstract**—Software controllers for physical processes are at the core of many safety-critical systems such as avionics, automotive engine control, and process control. Despite their importance, the design and implementation of software controllers remains an art form; dependability is generally poor, and the cost of verifying systems is prohibitive.

We illustrate the potential of applying program analysis tools on problems in controller design and implementation by focusing on concolic execution, a technique for systematic testing for software. In particular, we demonstrate how a concolic execution tool can be modified to automatically analyze controller implementations and (a) produce test cases achieving a coverage goal, (b) synthesize ranges for controller variables that can be used to allocate bits in a fixed-point implementation, and (c) verify robustness of an implementation under input uncertainties. We have implemented these algorithms on top of the Splat test generation tool and have carried out preliminary experiments on control software that demonstrates feasibility of the techniques.

## I. INTRODUCTION

Software implementations controlling physical systems form the cornerstone of many safety-critical systems, ranging from avionic and automotive control units to medical devices and power networks. For many of these systems, software is, unfortunately, the most brittle component; code quality is often poor and verification costs are high.

One reason is that the close interaction of computation with the physical world introduces a large semantic gap between specification and implementation. Consider the common practice of designing a control system using continuous mathematics and implementing it in software. At one end of the spectrum we have control engineering that studies the evolution of continuous dynamical systems while mostly ignoring the effect of implementing feedback control laws in software. At the other end, we have software engineering, which abstracts away physical resources when modeling computation. Thus, mathematical real numbers are approximated using floating-point or fixed-point arithmetic (losing many algebraic properties of the reals in the process), and the continuous world is discretized through sampling. These continuous and discrete abstractions are often incompatible, and efforts to combine them can cause incompatibilities at the interface.

One promising direction in bridging the gap is the adoption of *model-driven development* methodologies: instead

of directly coding the controller program, one starts with a domain-specific modeling language (such as Simulink) for the control system that allows the control engineer to specify the design at a level close to the mathematics, and generates the controller code automatically from this model. Verification and validation occurs at the model level, and the code is correct by construction.

Unfortunately, it is not so simple in practice. We do not have an adequate theory of cyber-physical design that can seamlessly transform the mathematical model into code. As a result, current design methodologies are burdened by unclear semantics at the modeling level, bugs in the code generator itself (which are often complicated pieces of code themselves), and ad hoc integration that tries to ensure the code implements the intended specification.

One missing piece in the design spectrum is the availability of program analysis tools that understand the semantics of the model on the one hand, and can work directly with source code on the other. Until a few years ago, and other than a few instances [1], [2], the worlds of program analysis and control design have not intersected to a large extent. We believe this is unfortunate. In particular, we believe recent progress in static and dynamic program analysis is likely to have a tremendous impact on reliable and cost-effective design of control systems.

We illustrate the applicability of program analysis tools on problems in controller design and implementation by focusing on Splat, a test generation tool for C programs [3]. Splat implements a concolic execution algorithm [4], [5] that runs the program simultaneously on concrete and symbolic values. Each (concrete) execution of the program generates an associated set of symbolic constraints. The solutions to these symbolic constraints gives new inputs that explore new execution paths of the program. By iteratively performing concolic execution and solving constraints, Splat provides a systematic testing algorithm that is, in the limit, guaranteed to provide full path coverage.

We describe our application of Splat on three problems arising out of controller design: path coverage, range analysis, and robustness of implementations. To understand these problems, let us first consider the workflow for controller design shown in Figure 1. Development of a control system starts with the design of a mathematical model of the physical system (or *plant*) to be controlled, typically as differential

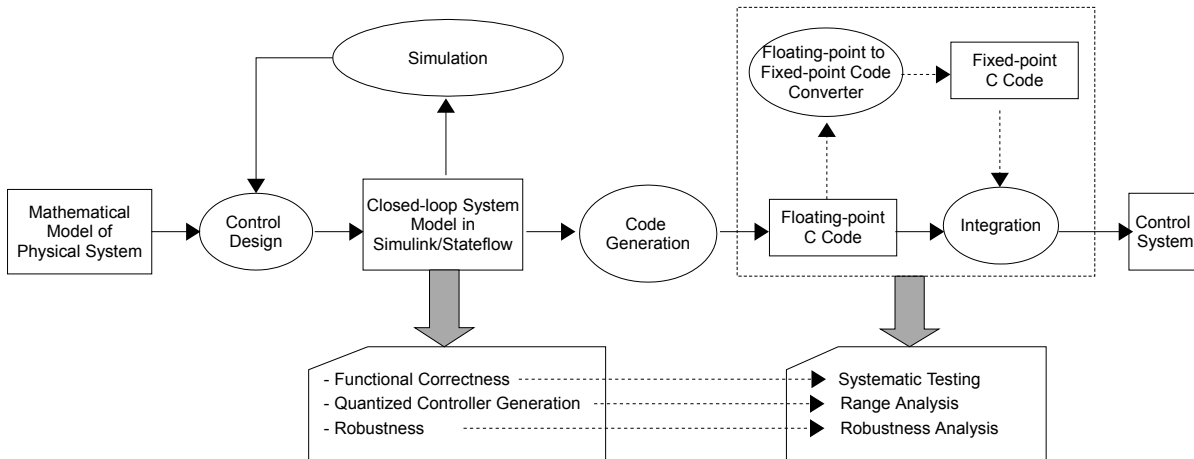


Figure 1. Control system design and implementation flow

equations whose solution gives the time evolution of the plant influenced by external inputs. Then control engineers design a *controller* that manipulates the external inputs to enforce a desired evolution of the plant. The behavior of the plant under the influence of the controller is then simulated with the help of modeling and simulation tools like Simulink/Stateflow to study the behavior of the system. If the simulation results are not satisfactory, the controller is modified or redesigned. The main goal at this stage is to ensure functional correctness, e.g., ensuring that the controller ensures stability of the system, and that small perturbations in the input only cause small deviations in the output (“robustness”). Once the control engineers are satisfied with the simulation results, the code for the controller is generated as either a floating-point or a fixed-point program. The generated code is again simulated, to make sure that the software implementation has not introduced unwanted jitter and that the desired performance of the controlled system is preserved.

Now consider the three program analysis problems. The first problem is “standard” in software development: find a test suite that exercises every path of the program (“path coverage”). Path coverage implies other coverage criteria such as branch or MC-DC coverage, which are often regulatory requirements on controller software [6]. Second, in order to generate a fixed-point program, we must find numerical ranges of all program variables (“range analysis”). The ranges are used to assign sufficiently many bits for the integer and fractional parts. Finally, in order to check robustness of the controller, we check, given the controller program, a metric on the input and output values, and a parameter  $\delta$ , what is the largest deviation in the outputs given a perturbation  $\delta$  in the input (“robustness analysis” [7]).

Generating test cases for control programs introduces some technical issues for concolic execution: controller implementations have extensive floating-point operations and non-linear constraints—features not supported by most concolic execution tools—and additionally, test generation has to take into account numerical stability of floating-point operations. We first extend Splat to add support for floating-point numbers and non-linear constraints. (The original version of Splat could only handle integer or Boolean symbolic variables.) To check for numerical instabilities in floating-point arithmetic, in our implementation, we abstract the concrete execution of the floating-point program using interval arithmetic [8], and use constraints over interval-valued variables in the symbolic execution. The interval computations are guaranteed to contain the real computation.

For range analysis and robustness checking, we modify the symbolic execution algorithm to get bounds on numerical ranges and on output deviations. Technically, for range analysis, instead of finding any solution satisfying the symbolic constraints for a path, we find the maximum and minimum values for each variable satisfying the symbolic constraints. For robustness checking, we consider symbolic constraints from pairs of program executions, and maximize the difference between the outputs in the two paths subject to the symbolic constraints as well as a constraint forcing the inputs to be within  $\delta$  [7].

We have implemented these algorithms in Splat, and we have applied our tool on a set of case studies involving control code. Our initial experimental results (outlined in Section V) demonstrate the applicability and usefulness of test generation tools in the domain of controller design.

## II. PROGRAM ANALYSIS PROBLEMS

We now describe the three problems (path coverage, range analysis, and robustness analysis) through examples.

```

void AC_output(float BlowerOut, float CompTorque,
              float EngineSpeed, float TcabinK)
{
    // local variable declarations
1   Maxflowrate = 0.12 * BlowerOut;
2   Product = EngineSpeed * CompTorque;
3   efficiency = 0.86 * Product;
4   assert(Maxflowrate != 0);
5   Product1 = efficiency / Maxflowrate;
   if (TcabinK < 220.0)
6       Tempenthalpy = TcabinK + 219.97;
   else
7       Tempenthalpy = 3.2029E+02 - TcabinK;
8   Sum = Tempenthalpy - Product1;
}

```

Figure 2. A simplified version of AC Control program

### A. Path Coverage

The path coverage problem takes as input a program  $P$  and produces as output a set  $T$  of program inputs (tests) that ensures that for each executable program path, there is a test  $t \in T$  that generates the program path. Complete path coverage increases confidence in the code by exercising each potential behavior, and by exhibiting possible assertion violations. Figure 2 shows a simplified version of C code generated from an AC control model available in Simulink Demo section [9]. There are three possible executions: the assertion failure on line 4, and the two branches in lines 6 and 7, assuming the assertion passes. Notice that the assertion prevents the potential divide-by-zero on line 5, by actively checking for the error condition. Without the assertion, there are two possible paths, but a test suite providing full path coverage can still miss the divide-by-zero error. Adding assertions to explicitly state program invariants before test generation is sometimes called *active checking* [10].

If the code has loops with potentially unbounded executions, the number of paths (and hence the size of the test suite demonstrating full path coverage) is unbounded. However, for control applications, most code have statically computed bounds for loops.

### B. Range Analysis

The *range* of a program variable  $v$  is defined as the least interval such that in every execution of the program, the value of  $v$  is in the interval. Generally for control programs, the ranges of the input variables are known. The range analysis problem takes as input a function and the ranges of the input variables of the function, and finds out the range of all the intermediate variables and output variables of the function. For example, the range analysis problem for the program in Figure 2 is as follows. Given ranges for the inputs BlowerOut, CompTorque, EngineSpeed, and TcabinK, if the function AC\_output is free from assertion failure, what are the ranges of all the assigned variables in line 1–7?

```

void Controller_output(float Desiredrpm, float N)
{
    // Local Variables declarations
1   rpmtorads = 0.1047 * Desiredrpm;
2   Sum = rpmtorads - N;
3   IntegralGain = 0.0723 * Sum;
4   RelOp1 = (float)(Sum <= 0.0);
5   RelOp2 = (float)(Sum >= 0.0);
6   LogicOp = (float)((RelOp1 != 0.0)
                    && (RelOp2 != 0.0));
   if (LogicOp >= 0.5)
7       Switch = IntegralGain;
   else
8       Switch = 0.0;
}

```

Figure 3. An adapted version of Engine Timing Controller program

### C. Robustness Analysis

Robustness is the property that small perturbations in the system inputs cause only small changes in its outputs. In robustness testing, for each input  $x_i$ , we need to find test vectors  $(v_1, \dots, v_n)$  and  $(v'_1, \dots, v'_n)$  that differ by at most  $\delta$  in the  $i$ th co-ordinate but are identical in all other co-ordinates (i.e.,  $v_j = v'_j$  for  $j \neq i$  and  $|v_i - v'_i| \leq \delta$ ) such that the difference in the outputs  $y$  and  $y'$  corresponding to the two inputs is maximized over all executions of the program.

Figure 3 shows a simplified version of Engine Timing Controller program. In this program, the inputs are Desiredrpm and N, and the output is Switch. In the robustness analysis problem, we would like to find out if there is a small deviation in the measurement of any of the inputs, what is the maximum possible deviation in the output Switch.

## III. CONCOLIC EXECUTION

We recapitulate the concolic execution algorithm [3]–[5], [11] on a simple imperative language.

### A. Programs

We represent a program  $P$  as a *control flow graph* (CFG)  $P = (X, X_0, \mathcal{L}, \ell_i, \text{op}, E)$  consisting of (1) a set of variables  $X$ , with a subset  $X_0 \subseteq X$  of *input* variables, (2) a set of control locations (or program counters)  $\mathcal{L}$  which includes a special start location  $\ell_i \in \mathcal{L}$ , (3) a function  $\text{op}$  labeling each location  $\ell \in \mathcal{L}$  with one of the following basic operations:

- an assignment  $x := e$ , where  $x \in X$  and  $e$  is an arithmetic expression over  $X$ ,
- a conditional  $\text{if}(e)\text{then } \ell' \text{ else } \ell''$ , where  $e$  is a side-effect free expression,
- an output statement  $\text{output}(y)$ , where  $y \in X$  is called the output variable,

and (4) a set of directed edges  $E \subseteq \mathcal{L} \times \mathcal{L}$  defined as follows. The set of edges  $E$  is the smallest set such that (i) every node  $\ell$  where  $\text{op}(\ell)$  is an assignment statement has exactly one node  $\ell'$  with  $(\ell, \ell') \in E$ , (ii) every node  $\ell$  such that  $\text{op}(\ell)$

is `if(e)then ℓ′ else ℓ″` has two edges  $(\ell, \ell')$  and  $(\ell, \ell'')$  in  $E$ . For a location  $\ell \in \mathcal{L}$  where  $\text{op}(\ell)$  is an assignment operation, we write  $N(\ell)$  for its unique neighbor.

Thus, the locations of a CFG correspond to program locations with associated commands, and edges correspond to control flow from one operation to the next. A *path* is a sequence of locations  $\ell^1, \ell^2 \dots \ell^n$  in the CFG such that  $(\ell^i, \ell^{i+1}) \in E$  for each  $1 \leq i < n$ .

The concrete semantics of the program is given using a *memory* that maps variables in  $X$  to values. For a memory  $M$ , we write  $M[x \mapsto v]$  for the memory mapping  $x$  to  $v$  and every other variable  $z \in X \setminus \{x\}$  to  $M(z)$ . We extend  $M$  to expressions  $e$  in the natural way:  $M(e)$  is the result of evaluating  $e$  using the memory  $M$ .

Execution starts from  $\ell_i$  with a memory  $M_0$  containing initial values for input variables in  $X_0$  and constant default values for variables in  $X \setminus X_0$ . Each operation updates the memory and the control location. Suppose the current location is  $\ell$  and the current memory is  $M$ . If  $\text{op}(\ell)$  is  $x := e$ , then the new location is  $N(\ell)$  and the new memory is  $M[x \mapsto M(e)]$ . If  $\text{op}(\ell)$  is `if(e)then ℓ′ else ℓ″`, then the new location is  $\ell'$  if  $M(e) \neq 0$  and is  $\ell''$  if  $M(e) = 0$ . In either case, the memory remains unchanged. If  $\text{op}(\ell)$  is the output statement  $\text{output}(y)$ , then the program terminates with output  $M(y)$ .

Execution of the program starting from a memory  $M_0$  defines a path in the CFG in a natural way. A path is executable if it is the path corresponding to program execution from some initial memory  $M_0$ .

## B. Concolic Execution

In concolic execution, the program is executed simultaneously on concrete and symbolic inputs. The symbolic constraints produced during a run are used to select inputs that follow a different program execution. The concolic execution algorithm executes the program while maintaining two additional artifacts: a *symbolic memory*  $\mu$  which maps variables in  $X$  to symbolic expressions over a set of symbolic constants, and a *path constraint*  $\xi$ , which collects predicates over symbolic constants along the execution path. The symbolic memory map and the symbolic path constraint are updated during the course of execution.

Execution proceeds as in the concrete case, starting at  $\ell_i$  with the initial memory  $M_0$ . Initially, the symbolic memory  $\mu$  maps each input variable  $x \in X_0$  to a fresh symbolic constant  $\alpha_x$  and each variable  $z \in X \setminus X_0$  to some default constant value. Initially, the path constraint is *true*.

For an assignment  $x := e$ , the concrete memory is updated as in the concrete semantics. The symbolic memory  $\mu$  is updated to  $\mu[x \mapsto \mu(e)]$ , where  $\mu(e)$  denotes the symbolic expression obtained by evaluating  $e$  using the map  $\mu$ . The path constraint is unchanged. The control location is updated to  $N(\ell)$ .

For a conditional `if(e)then ℓ′ else ℓ″`, there is a choice in updating the control location. If the concrete execution goes to the control location  $\ell'$  (i.e., if  $M(e) \neq 0$ ), the path constraint is updated to  $\xi \wedge \mu(e) \neq 0$ . Otherwise, if the new control location is  $\ell''$  (i.e., if  $M(e) = 0$ ), the path constraint is updated to  $\xi \wedge \mu(e) = 0$ . In each case, the new symbolic memory is still  $\mu$ .

At the end of an execution, a new execution is created by selecting a conditional `if(e)then ℓ′ else ℓ″` along the path that was executed such that (1) the current execution took the *then* (respectively, *else*) branch of the conditional, and (2) the path that agrees with the current execution up to  $\ell$  but then takes the *else* (respectively, *then*) branch of this conditional has not been explored before. Let  $\mu_\ell$  and  $\xi_\ell$  be the symbolic memory and path constraint just before executing this conditional. We find a satisfying assignment for the constraint  $\xi_\ell \wedge \mu_\ell(e) = 0$  (respectively,  $\xi_\ell \wedge \mu_\ell(e) \neq 0$ ). The property of a satisfying assignment is that if these inputs are provided at the input statements, then the new execution will follow the old execution up to the conditional at  $\ell$ , but then take the conditional branch opposite to the one taken by the old execution, thus ensuring that the other branch is covered. The satisfying assignment is used to define a new input for the next execution of the program. In this way, each path of the program can be traversed. A satisfying assignment for these constraints can be computed by invoking a decision procedure.

## IV. ALGORITHMS

We now show how the concolic execution algorithm, with small modifications, can be used to solve the program analysis questions from Section II.

### A. Path Coverage

1) *Problem Definition:* The path coverage problem takes as input a program  $P$  and asks to generate a set  $T$  of initial memories such that for each executable path  $\pi$  of  $P$  there is an initial memory  $M_0 \in T$  such that execution of  $P$  with input  $M_0$  generates path  $\pi$ . If  $P$  does not have loops, the set  $T$  is guaranteed to be finite.

2) *Algorithm Outline:* We use “basic” concolic execution to implement path coverage. One possibility is to model each floating-point variable and each floating-point operation in the program in a bit-precise way, using the IEEE 754 standard for floating-points, and to reduce reasoning about floating-point numeric operations using a Boolean satisfiability solver (see [12]). Our experience is that it is hard to scale this encoding, especially in the presence of several non-linear operations.

Instead, we take the following route. Instead of performing the concrete execution based on floating-point numbers, we instrument the program to change each floating-point variable to an interval-valued variable, and change each

floating-point operation to the corresponding interval arithmetic operation. Interval arithmetic has the property that the intervals are guaranteed to contain the precise result of the computation [8]. After the transformation, a memory  $M$  maps each variable  $x$  to an interval  $i$  that contains the corresponding floating-point value  $v$  for the variable. The symbolic memories and path constraints generated during concolic execution are also interpreted over intervals. We use a decision procedure for non-linear interval-valued constraints which performs a numerical search and provides satisfying assignments as numerical bounds on intervals with the property that a satisfying assignment is guaranteed to exist within the given intervals [13]. As our concrete execution is based on intervals, the solution of the path constraint can be directly fed to the program as the input for the next execution.

One problem with the approach is that the interval bounds may “stride” a conditional (called “lack of robustness” in [14]), i.e., a memory can satisfy both a conditional expression and its negation. For example, if  $e$  is of the form  $x > 1.0$ , and in the current memory  $M$ , we have  $M(x) = [0.99, 1.01]$ , then it is not clear if the *then* or the *else* branch should be taken. In this situation we say that  $e$  is evaluated to a special value “don’t know”, and our algorithm raises an exception.

3) *Example:* In the AC control code in Figure 2, at line 4 there is a possibility of divide by zero error. The assert statement ensures that the program does not reach statement 4 with  $\text{Maxflowrate} = 0$ . From exhaustive testing of the program we can find out that if the range of  $\text{BlowerOut}$  does not include 0, then the divide-by-zero bug does not arise.

## B. Range Analysis

1) *Problem Definition:* Let  $P = (X, X_0, \mathcal{L}, \ell_i, \text{op}, E)$  be a program. Suppose that for each input variable  $x \in X_0$ , we are given a pre-condition  $l_x \leq x \leq u_x$ , where  $l_x, u_x \in \mathbb{R}$ , restricting the value of  $x$  in the initial memory. The *range* of a variable  $v \in X$  is the least interval  $[l_v, u_v]$ , such that for any execution of  $P$  with the initial memory satisfying  $l_x \leq x \leq u_x$  for each  $x \in X_0$ , we have that the value of  $v$  is always in the interval  $[l_v, u_v]$ . The *range analysis problem* takes as input a program  $P$  and a pre-condition mapping each input variable to an interval, and returns a mapping from variables in  $X$  to their ranges.

2) *Algorithm Outline:* We now show how the concolic execution algorithm can be slightly modified to compute ranges. We assume that the programmer provides ranges for the input variables. Intuitively, for each variable  $v \in X$ , whenever  $v$  is assigned to during concolic execution, we perform two optimization problems to find the maximum and minimum values of  $\mu(v)$  given the current path constraint  $\xi$ . The range for  $v$  is  $[l_v, u_v]$ , where  $l_v$  (respectively,  $u_v$ ) is

the infimum (resp., supremum) of all the minimum (resp., maximum) values over all executed paths.

3) *Example:* We illustrate our range analysis technique on the AC control code shown in Figure 2. Initially, we know only the ranges of the input variables  $\text{BlowerOut}$ ,  $\text{CompTorque}$ ,  $\text{EngineSpeed}$ , and  $\text{TcabinK}$ . For simplicity, let us assume that the range of each input variable is  $[1, 10]$ . Let the symbolic values for the four inputs be  $\alpha_1, \alpha_2, \alpha_3$ , and  $\alpha_4$ , respectively. There are two paths in the program. During the execution of the first path, we compute the range for  $\text{Maxflowrate}$ ,  $\text{Product}$ ,  $\text{efficiency}$ ,  $\text{Product1}$ ,  $\text{Tempenthalpy}$  at line 5 and  $\text{Sum}$ . Each range computation requires solving a minimization and a maximization problem. For example, the minimization problem to compute the range of  $\text{Maxflowrate}$  is

$$\begin{aligned} \min & 0.12 \cdot \alpha_1 \\ \text{s.t.} & \alpha_1 \geq 1 \wedge \alpha_1 \leq 10 \end{aligned}$$

Similarly, the maximization problem to be solved to compute the range of  $\text{Sum}$  is as follows

$$\begin{aligned} \max & (\alpha_4 + 219.97) - (0.86 \cdot \alpha_3 \cdot \alpha_2) / (0.12 \cdot \alpha_1) \\ \text{s.t.} & \alpha_1 \geq 1 \wedge \alpha_1 \leq 10 \wedge \alpha_2 \geq 1 \wedge \alpha_2 \leq 10 \wedge \\ & \alpha_3 \geq 1 \wedge \alpha_3 \leq 10 \wedge \\ & \alpha_4 \geq 1 \wedge \alpha_4 \leq 10 \wedge \alpha_4 < 220.0 \end{aligned}$$

Using our algorithm we find the ranges for the variables of the function in Figure 2. The ranges for  $\text{Maxflowrate}$ ,  $\text{Product}$ ,  $\text{efficiency}$ ,  $\text{Product1}$ ,  $\text{Tempenthalpy}$  and  $\text{Sum}$  are  $[0.120000, 1.200000]$ ,  $[1.000000, 100.000000]$ ,  $[0.860000, 86.000000]$ ,  $[0.716667, 716.666667]$ ,  $[220.970001, 229.970001]$  and  $[-495.696666, 229.253334]$  respectively.

4) *Compositional Analysis:* The range analysis above does not scale well, as the size of the constraints needed to be solved at the end of each execution increases with path size. Therefore, we implement a *compositional* algorithm based on function summaries [15], in which range analysis of a function is done independent of its callers. For each function, we maintain a set of *summaries*, where each summary relates the ranges of the inputs to the function with the range of the output. During concolic execution if we encounter a function call of the form  $y = f(\text{par}_1, \dots, \text{par}_n)$ , based on the current ranges of the parameters, we compute the ranges of all assigned variables in the called function. The range of the returned variable is used to update the range of  $y$ .<sup>1</sup>

The program in Figure 4 is an extended version of the program presented in Figure 2. The range for the left hand side variables in the assignments in line 1 to line 7 are

<sup>1</sup>Our presentation of concolic execution in Section III omits functions and function calls. However, our algorithm can be extended to handle function calls (see [4], [15]).

```

float lin_search(float input_value)
{
    float output_value;
    if (input_value < 219.97)
10     output_value = 220.0;
    else
11     output_value = 320.0;
    return output_value;
}

void AC_output(float BlowerOut, float CompTorque,
               float EngineSpeed, float TcabinK)
{
    // local variables declarations
1  Maxflowrate = 0.12 * BlowerOut;
2  Product = EngineSpeed * CompTorque;
3  efficiency = 0.86 * Product;
4  Product1 = efficiency / Maxflowrate;
   if (TcabinK < 220.0)
5     Tempenthalpy = TcabinK + 219.97;
   else
6     Tempenthalpy = 3.2029E+02 - TcabinK;
7  Sum = TcabinK - Product1;
8  ExitTempAC = lin_search(Sum);
}

```

Figure 4. An example to illustrate compositional range analysis of a program

calculated in the same way as before. In line 8 there is a function call. In naive range analysis, we can inline the function body, and compute the range of the assigned variables in the called function in the same way as before. However, when we visit the function call `lin_search` we know the range of its input parameter `Sum`. Thus we can compute the ranges of the assigned variables in `lin_search` function independently based on the ranges of its input parameter. Once we are done with range computation in `lin_search` function, the range of the returned variable `output_value` becomes the range of the variable `Sum` in `AC_output` function.

Further, we can tradeoff precision and scalability by *abstracting* function summaries, e.g., by ignoring any correlation between the input parameters. This enables a simple representation of summaries as tuples of intervals, but may lead to loss of precision. Figure 5 illustrates this situation. The function `sin_to_tan` takes the sine of an angle and returns the tangent of the angle. Let us assume that the range of input `x` of the `sin_to_tan` function is  $[-0.5, 1]$ , which corresponds to a range of angle  $[-\frac{\pi}{4}, \frac{\pi}{2}]$ . The range of `y1` is  $[0, 1]$ , and so is the range of `y2`. The function `sin_to_tan` calls the function `ratio` with arguments `x` and `y2`. The range of `z` in the function `ratio` is  $(-\text{Inf}, \text{Inf})$ , which becomes the range of `result` in `sin_to_tan`. Thus we get a range  $(-\text{Inf}, \text{Inf})$  for the tangent of the angle with range  $[-\frac{\pi}{4}, \frac{\pi}{2}]$ . However, for this range of the angle we should get the range of `result` to be  $[-1, \text{Inf})$ . The range is sound, as the computed range includes the actual range, but imprecise. To deal with this situation, our implementation performs

```

double ratio (double x, double y)
{
    double z;
    if (y == 0)
        if (x < 0)
            z = -Inf;
        else
            z = Inf;
    else
        z = x / y;
    return z;
}

double sin_to_tan (double x)
{
    double y1, y2;
    double result;
    y1 = 1 - x * x;
    y2 = sqrt (y1);
    result = ratio (x, y2);
    return result;
}

```

Figure 5. An example to illustrate that compositional range analysis may provide inaccurate result

a dataflow analysis to detect if there is any correlation between the actual parameters of a function call. If there is a correlation, then we do not abstract the summary information.

### C. Robustness Analysis

1) *Problem Definition:* Let  $P = (X, X_0, \mathcal{L}, \ell_i, \text{op}, E)$  be a program. We assume for simplicity that there is exactly one node  $\ell_o$  labeled `output(y)`, and each program execution terminates at  $\ell_o$ . We call the variable  $y \in X$  the *output variable*. Let us fix an input  $x \in X_0$ . Let  $\delta_x$  denote the maximum possible uncertainty in measuring input  $x$ . Measurement errors in  $x$  can cause the output of the program to change in two possible ways: (i) either the program executes along some new path due to a change in the result of a conditional dependent on  $x$ , (ii) or the program executes along the same path but the output variable depends on the value of  $x$ . We define the *maximum output sensitivity* [7] of  $y$  w.r.t.  $x$  and  $\delta_x$  as follows:

$$\delta_{yx} = \max_{\bar{v}, x, x'} \left\{ |y - y'| \mid \begin{array}{l} y = P(\bar{v}, x) \\ y' = P(\bar{v}, x') \\ |x - x'| \leq \delta_x \end{array} \right\}$$

where  $x$  and  $x'$  denote the actual and measured value of  $x$  respectively,  $\bar{v}$  is the value of all other input variables, and  $y$  and  $y'$  denote the value of the output of the program for input  $\bar{v}, x$  and  $\bar{v}, x'$  respectively. Informally,  $\delta_{yx}$  is the maximum possible change in the value of the output  $y$  when in the input of the program, the variable  $x$  deviates by at most  $\delta_x$ . A program  $P$  is  $(\delta, \epsilon)$ -robust with respect to an input  $x \in X_0$  if  $\delta_{yx} \leq \epsilon$  whenever  $\delta_x \leq \delta$ .

2) *Algorithm Outline:* To find the maximum sensitivity of output  $y$  of a program with respect to an input  $x$ , we

perform concolic execution on the program and collect, for each execution of the program, a pair  $\langle e, \xi \rangle$  of the value of  $y$  in the symbolic memory on termination and the path constraint for the path at termination. Then for each pair  $\langle e_1, \xi_1 \rangle$  and  $\langle e_2, \xi_2 \rangle$ , we find out the maximum deviation in output  $y$  when the input  $x$  is bounded by  $\delta_x$  and the two execution paths are assumed to satisfy  $\xi_1$  and  $\xi_2$ . This is done by formulating and solving a maximization problem as described below.

For a symbolic expression  $e$ , let the *primed version*  $e'$  denote the symbolic expression in which every variable  $\alpha$  in  $e$  is replaced by a primed copy  $\alpha'$ . Similarly, for a path constraint  $C$ , let the *primed version*  $C'$  denote the same constraint in which every variable  $\alpha$  is replaced by a primed copy  $\alpha'$ . For example, the primed version of  $x + y$  is  $x' + y'$  and the primed version of  $x \geq 0 \wedge y = x$  is  $x' \geq 0 \wedge y' = x'$ . The maximization problem is now defined as follows. The objective function is  $|e_1 - e_2'|$ , which represents the difference between the symbolic outputs along the two paths (note that  $e_2'$  is primed). The constraints of the optimization problem are

$$\xi_1 \wedge \xi_2' \wedge \bigwedge_{z \in X_0 \setminus \{x\}} \alpha_z = \alpha'_z \wedge |\alpha_x - \alpha'_x| \leq \delta_x \quad (1)$$

The first two conjuncts enforce that the two execution paths follow the path constraints  $\xi_1$  and  $\xi_2$  respectively, the third conjunct  $\bigwedge_{z \in X_0 \setminus \{x\}} \alpha_z = \alpha'_z$  enforces every input variable other than  $x$  to be equal on the two executions, and the final constraint  $|\alpha_x - \alpha'_x| \leq \delta_x$  ensures that the deviation in the input  $x$  is bounded by  $\delta_x$ .

Thus, the optimization problem is:

$$\begin{aligned} \max \quad & |e_1 - e_2'| \\ \text{s.t.} \quad & \text{constraints (1)} \end{aligned}$$

The optimization problem above may not always be satisfiable. If that is the case, we know that when the maximum deviation of input  $x$  is bounded by  $\delta_x$ , the program execution cannot change from one to the other in the path pair. Otherwise, the value of the objective function gives us the maximum possible deviation in the output for the path pair under consideration. Finally, the maximum output sensitivity for  $y$  w.r.t. input  $x$  is defined as the maximum deviation over all path pairs.

3) *Example*: Let us illustrate our robustness analysis algorithm on the Engine Timing Controller Example in Figure 3. Based on the value of Sum, the program has three possible paths. Let the symbolic values for input Desiredrpm and N be  $\alpha_1$  and  $\alpha_2$  respectively. The output variable for the program is Switch. The symbolic expressions and path constraints for the three paths are as follows:

$$\begin{aligned} \text{Path 1: Symbolic Output Expression:} \\ & 0.0723 \cdot (0.1047 \cdot \alpha_1 - \alpha_2) \\ \text{Path Constraint: } & 0.1047 \cdot \alpha_1 - \alpha_2 == 0 \end{aligned}$$

$$\begin{aligned} \text{Path 2: Symbolic Output Expression: } & 0.0 \\ \text{Path Constraint: } & 0.1047 \cdot \alpha_1 - \alpha_2 > 0 \\ \text{Path 3: Symbolic Output Expression: } & 0.0 \\ \text{Path Constraint: } & 0.1047 \cdot \alpha_1 - \alpha_2 < 0 \end{aligned}$$

In this example there are three path pairs. For the path pair (Path2, Path3), the outputs are concrete and the same. So, due to small deviation in the input the output does not change. Careful analysis also reveals that the other two path pairs are also not sensitive to any of the inputs. Thus, in the function in Figure 3, the output is constant. (Notice that the value of Switch is zero in all cases.) However, if the line 8 in Figure 3 was `Switch = C` instead of `Switch = 0.0`, then for each  $\delta > 0$ , the output could deviate by  $C$  units when the input is perturbed by  $\delta$ .

## V. IMPLEMENTATION

### A. Tool Architecture

Figure 6 shows the architecture of our tool. The core of the tool is a concolic execution engine which uses a constraint solver, and is interfaced with a nonlinear optimization tool to support solving optimization problems required for the analysis. We use Splat [3], [16] as the basis for our concolic execution tool and we use CIL [17] to instrument programs.

We use a combination of HySAT [13] and Yices [18] to solve mixed integer-floating-point non-linear constraints arising out of symbolic execution. HySAT provides interval bounds for floating-point variables, which are used as inputs in our interval-based test generation. To facilitate interval-based computation during the concrete runs, we use the multi-precision interval arithmetic library MPFI [19]. We modify Splat's instrumentation to modify each floating-point variable in the program with an interval-valued variable, and to use interval-arithmetic operations instead of floating-point operations.

We use the Lindo optimization API [20], which provides general nonlinear and nonlinear/integer optimization problem solving APIs, for solving the optimization problems. As the optimization problems are in general non-linear, Lindo may not always produce a globally optimal result. If Lindo indicates that a solution is local, we use HySAT to iteratively find out the global optimum value as presented in [21]

For our case studies, we either use Simulink models from which we generate C code using Real-Time Workshop or use code written originally in C.

### B. Case Studies

We now describe a few case studies that we have carried out on different control systems. These systems include a train car controller [22] and a set of examples from the Simulink demo section [9], e.g., an AC controller in an automatic climate control system, an engine timing controller, and a fuel rate calculator in a fault-tolerant fuel control system.

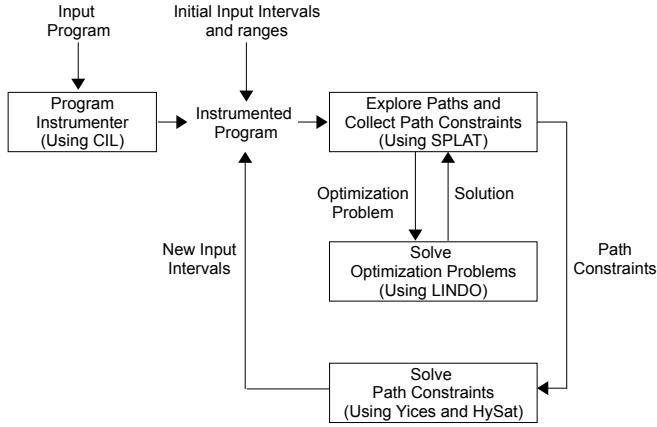


Figure 6. Control Program Analysis Tool Architecture

The train car controller is used in a locomotive pulling a train car to quickly reduce the oscillations between the locomotive and the car. The controller is linear, and its software implementation involves a series of addition and multiplication operations. Depending on how the arithmetic operations are scheduled in the program, the ranges of the intermediate variables vary. Starting with the controller program generated by Real-time Workshop [23] we manually changed the order of the arithmetic operations using the rules of commutativity, associativity, and distributivity to get different versions of the same controller. From our experiment, we found that it is possible to find a controller implementation which is more optimized than the controller program generated by the automatic code generator in terms of ranges of the intermediate variables. This helps us get a better fixed-point implementation for the controller.

The AC controller system controls the temperature of the air that exits from the A/C of the car based on the internal temperature of the car and engine speed and compressor torque. First, the inputs are used to perform some nonlinear computations and then the result is used to find out the value of the output from a lookup table. This is very typical in nonlinear control system implementations in automotive domain and existing test generation tools are very unlikely to achieve full path coverage on this kind of control code. Our tool finds out a possibility of divide by zero error in the code for the ranges of the inputs chosen based on Simulink simulation of the model and once we choose the input ranges suitably to avoid the divide by zero error, we can successfully achieve full path coverage by exploring all 133 paths in the program.

The engine time controller implements a PI controller [24] that regulates engine speed with a fast throttle actuator. PI controllers or more generally PID controllers [24] are used where the controller has to be oblivious of the underlying process. A PID controller tries to minimize the difference

between a measured process variable and a desired set point by adjusting the process control inputs. In a PID controller the integral value determines the reaction of the controller based on the sum of recent errors. To achieve this the output of the controller is fed back as an input to the controller, and the sum of the error is calculated in a loop. This creates a problem in analyzing the code generated for PID controllers as the loop cannot be unrolled statically. However, the output of the controller is limited by saturation, and that plays as a remedy of the problem in analyzing the control code for PID controller. We consider the output of the controller as a new input which has range defined by the saturation limits of the output. In this way we get rid of the loop and our results still remain sound. We cannot achieve full path coverage on this program, as during concolic execution, a conditional is evaluated to “don’t know” value. This implies that due to small change in the input the program may be executed along two different paths. This is an indication of loss of robustness. However, as we cannot get full path coverage, we cannot find out the robustness of the program quantitatively, and the indicated loss of robustness may actually be a false alarm.

The fuel rate calculator is the part of a fault-tolerant fuel control system that maintains the optimal air-fuel ratio in the automobile engine. The fuel rate calculator calculates the fuel rate depending on the fuel mode. This is an example of switching control system [25] whose configuration changes at runtime depending on the mode of the system. The fuel rate calculator has two subsystems: feedforward fuel rate calculator and switchable compensator. For range analysis, we employ both compositional and non-compositional techniques and achieve full path coverage in both the cases. The non-compositional technique calculates precise ranges, but takes 22% more time than the compositional technique. The compositional technique, though faster than the non-compositional one, makes the range results imprecise, as we ignore the relationship between the input parameters that are required to give precise ranges for the switchable compensator subsystem. The ranges of all the variables in the function implementing feedforward fuel rate calculator subsystem are same for both non-compositional and compositional analysis as the parameters of the function are independent. But for the function implementing the switchable compensator, there are data dependencies among the input parameters that results in imprecision in the range of the variables in the function when calculated in a compositional manner. For example, the range for the return value of the function is computed to be  $[-0.2592, 1.7883]$  in the non-compositional case, but  $[-0.2796, 2.1993]$  in the compositional analysis. We also compared our result with an interval based abstract interpretation tool, the online interface of the Interproc Analyzer [26] with the *Box* numerical abstract domain. The range for the return value for the switchable compensator function as computed by Interproc is  $[-0.5440,$

2.2944], which is less precise than our result obtained by compositional analysis.

Further experimental results on robustness analysis can be found in [7].

## VI. RELATED WORK

Program analysis techniques (based on abstract interpretation) have been applied to statically prove properties of real-time control programs, the most notable tools being Astrée [1] and Fluctuat [2], [27]. Both tools perform sophisticated static analysis to reason about numerical errors in computations, as well as the absence of runtime errors such as division-by-zero. They have been successfully applied to avionics code. The focus in these tools is on proving programs correct rather than generating tests. Constraint solving for non-linear constraints is side-stepped using cleverly designed linear abstractions that do not lose too much precision.

Systematic testing of programs based on concolic execution [4], [5], [15], [16] has received considerable attention and industrial success. Most previous works and implementations focused on systems code which did not have floating-point variables, or for which the floating-point variables could be ignored [28]. Kanade et al. [29] presented a concolic execution based technique for systematic exploration of distinct behaviors of a Simulink/Stateflow model containing only linear Simulink blocks. Satpathy, Yeolekar, and Ramesh [30] applied concolic execution for Simulink/Stateflow models, and used pattern-guided heuristics for tackling nonlinear blocks. In contrast, we use a decision procedure for solving non-linear constraints based on interval semantics. Numerical decision procedures for solving non-linear constraints based on interval search have been used to generate test cases for programs from the automotive and avionics domains from predicate-abstraction-based reachability analysis [31], [32]. Fainekos et al. [14] also consider a framework for determining the numerical robustness of simulations of hybrid systems against floating-point rounding errors and system modeling uncertainties, based on interval and affine arithmetic. While our current decision procedure does not precisely model the IEEE 754 floating point standard, there has been several attempts at decision procedures modeling the standard [12], [33].

The range analysis problem has been studied extensively in the context of optimum bit-width allocation to intermediate variables in a fixed-point program, mostly in the DSP domain. Both static [34]–[36] and simulation-based [37], [38] approaches have been used. Static approaches are usually based on interval arithmetic [8] or affine arithmetic [39]. However, in the presence of non-affine operations like multiplication and division, both techniques can be imprecise. Though this phenomena is rare in DSP applications [40], in control programs, presence of a series of multiplication and division, as well as multiple branching, is very common.

Static analyzers based on abstract interpretation [41], though fast, suffer from the limited expressiveness and lack of precision in the result. Simulation-based methods, especially those performing constrained-random simulations, suffer from lack of completeness. This causes the resultant system to be non-robust, incomplete simulation can lead to overflow conditions resulting in incorrect behavior. Our compositional approach combines simulation (using concolic execution) with static techniques (summarization). Recently Kinsman and Nicolici [21] proposed a method of range refinement for scientific computing based on decision procedures, but focused only on branch-free code fragments.

The problem of checking robustness of software implementations was studied in the context of control applications in [7], and independently, for general software applications in [42].

## VII. CONCLUSION

We believe controller implementations form an interesting niche application for static and dynamic program analysis tools. The applications in this domain have a high cost of failure. At the same time, controller software, for the most part, does not have many characteristics of general-purpose application software (such as statically unbounded loops or recursion, complex pointer-based data structures, etc.) which make the application of program analysis difficult in the general case.

While we have demonstrated the potential of applying systematic testing tools in the design of controllers, our applications so far have been “low level” properties of code. We envision that in the future, program analysis tools can also be used to reason about “higher-level” properties of the application. For example, while the control engineer proves stability of the mathematical model of the controlled system, it is unclear whether the implemented controller still guarantees stability, and it will be useful to have automatic tools that can reason about stability of the implementation.

**Acknowledgments.** This research was funded in part by the NSF awards CCF-0546170 and CNS-0720881. Zilong Wang’s stay at UCLA was partly funded by a CSST scholarship.

## REFERENCES

- [1] 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 *Proc. PLDI*, 2003, pp. 196–207.
- [2] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védryne, “Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software,” in *Proc. FMICS*, 2009, pp. 53–69.
- [3] R. Xu, “Symbolic Execution Algorithms for Test Generation,” Ph.D. dissertation, University of California, Los Angeles, 2009.
- [4] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed Automated Random Testing,” in *Proc. PLDI*, 2005, pp. 213–223.

- [5] K. Sen, D. Marinov, and G. Agha, "CUTE: A Cconcolic Unit Testing Engine for C," in *Proc. FSE*, 2005, pp. 263–272.
- [6] RTCA SC-167/EUROCAE WG-12, "DO178B/ED12B - Software Considerations in Airborne Systems and Equipment Certification, Washington D.C., RTCA Inc." 1992.
- [7] R. Majumdar and I. Saha, "Symbolic Robustness Analysis," in *Proc. Real-Time Systems Symposium*, 2009, pp. 355–364.
- [8] R. Moore, *Interval Analysis*. Prentice Hall, 1966.
- [9] The MathWorks, "Simulink 7.5 Demos," <http://www.mathworks.com/products/simulink/demos.html>.
- [10] P. Godefroid, M. Levin, and D. Molnar, "Active Property Checking," in *Proc. EMSOFT 08*, 2008, pp. 207–216.
- [11] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically Generating Inputs of Death," in *Proc. CCS*, 2006, pp. 322–335.
- [12] A. Brillout, D. Kroening, and T. Wahl, "Mixed Abstractions for Floating-point Arithmetic," in *Proc. FMCAD 09*, 2009, pp. 69–76.
- [13] M. Fränzle and C. Herde, "HySAT: An Efficient Proof Engine for Bounded Model Checking of Hybrid Systems," *Formal Methods in System Design*, vol. 30, no. 3, pp. 179–198, 2007.
- [14] G. Fainekos, S. Sankaranarayanan, F. Ivancic, and A. Gupta, "Robustness of Model-Based Simulations," in *Proc. Real-Time Systems Symposium*, 2009, pp. 345–354.
- [15] P. Godefroid, "Compositional Dynamic Test Generation," in *Proc. POPL*, 2007, pp. 47–54.
- [16] R. Xu, P. Godefroid, and R. Majumdar, "Testing for Buffer Overflows with Length Abstraction," in *Proc. ISSSTA*, 2008, pp. 27–38.
- [17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Proc. CC*, 2002, pp. 213–228.
- [18] SRI International, "Yices: An SMT Solver," <http://yices.csl.sri.com/>.
- [19] F. Rouillier and N. Revol, "MPFI 1.0," <http://perso.ens-lyon.fr/nathalie.revol/mpfi.html>.
- [20] LINDO Systems Inc., "LINDO API 6.0 - Powerful Library of Optimization Solvers and Mathematical Programming Tools," <http://www.lindo.com/>.
- [21] A. B. Kinsman and N. Nicolici, "Finite Precision Bit-width Allocation Using SAT-Modulo Theory," in *Proc. DATE*, 2009, pp. 1106–1111.
- [22] P. McLane, L. Peppard, and K. Sundareswaran, "Decentralized Feedback Controls for the Brakeless Operation of Multilocomotive Powered Trains," *IEEE Transactions on Automatic Control*, vol. 21, no. 3, pp. 358–363, 1976.
- [23] The MathWorks, "Real-Time Workshop 7.5," <http://www.mathworks.com/products/rtw/>.
- [24] Y. Li, K. H. Ang, and G. Chong, "PID Control System Analysis and Design," *IEEE Control Systems Magazine*, pp. 559–576, 2006.
- [25] D. Liberzon, *Switching in Systems and Control*. Birkhäuser, 2003.
- [26] "The Interproc Analyzer," <http://pop-art.inrialpes.fr/interproc/interprocweb.cgi>.
- [27] O. Bouissou, E. Goubault, S. Putot, K. Tekkal, and F. Védrine, "HybridFluctuat: A Static Analyzer of Numerical Programs within a Continuous Environment," in *Proc. CAV*, 2009, pp. 620–626.
- [28] P. Godefroid and J. Kinder, "Proving Memory Safety of Floating-Point Computations by Combining Static and Dynamic Program Analysis," in *Proc. ISSSTA*, 2010.
- [29] A. Kanade, R. Alur, F. Ivancic, S. Ramesh, S. Sankaranarayanan, and K. C. Shashidhar, "Generating and Analyzing Symbolic Traces of Simulink/Stateflow Models," in *Proc. CAV*, 2009, pp. 430–445.
- [30] M. Satpathy, A. Yeolekar, and S. Ramesh, "Randomized Directed Testing (REDIRECT) for Simulink/Stateflow Models," in *Proc. EMSOFT*, 2008, pp. 217–226.
- [31] S. Xia, B. D. Vito, and C. Muñoz, "Towards Automated Test Generation for Engineering Applications," in *Proc. ASE*, 2005, pp. 283–286.
- [32] S. Xia, B. D. Vito, and C. Muñoz, "Predicate Abstraction of Programs with Non-linear Computation," in *Proc. ATVA*, 2006, pp. 352–368.
- [33] B. Botella and C. Gotlieb, "Symbolic Execution of Floating-point Computations," *Softw. Test., Verif. Reliab.*, vol. 16, pp. 97–121, 2006.
- [34] D. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-Guaranteed Bit-width Optimization," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–2000, 2006.
- [35] J. A. López, C. Carreras, and O. Nieto-Taladriz, "Improved Interval-based Characterization of Fixed-point LTI Systems with Feedback Loops," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 11, pp. 1923–1932, 2007.
- [36] W. G. Osborne, R. C. C. Cheung, J. G. F. Coutinho, W. Luk, and O. Mencer, "Automatic Accuracy-Guaranteed Bit-width Optimization for Fixed and Floating-point Systems," in *Proc. FPL*, 2007, pp. 617–620.
- [37] P. Belanovic and M. Rupp, "Automated Floating-point to Fixed-point Conversion with the Fixify Environment," in *Proc. International Workshop on Rapid System Prototyping*, 2005, pp. 172–178.
- [38] A. Mallik, D. Sinha, P. Banerjee, and H. Zhou, "Low-Power Optimization by Smart Bit-width Allocation in a SystemC-based ASIC Design Environment," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 3, pp. 447–455, 2007.
- [39] J. Stolfi and L. H. Figueiredo, "Self-validated Numerical Methods and Applications," in *Monograph for 21st Brazilian Mathematics Colloquium, Rio de Janeiro: IMPA*, 1997.
- [40] C. F. Fang, R. A. Rutenbar, and T. Chen, "Fast, Accurate Static Analysis for Fixed-point Finite-precision Effects in DSP Designs," in *Proc. ICCAD*, 2003, pp. 275–282.
- [41] B. Jeannot and A. Miné, "Apron: A Library of Numerical Abstract Domains for Static Analysis," in *Proc. CAV*, 2009, pp. 661–667.
- [42] S. Chaudhuri, S. Gulwani, and R. Lublinerman, "Continuity Analysis of Programs," in *Proc. POPL*, 2010, pp. 57–70.