Symbolic Robustness Analysis

Rupak Majumdar
Department of Computer Science
University of California, Los Angeles
CA 90024, USA
Email: rupak@cs.ucla.edu

Indranil Saha
Department of Computer Science
University of California, Los Angeles
CA 90024, USA
Email: indranil@cs.ucla.edu

***Abstract*—A key feature of control systems is *robustness*, the property that small perturbations in the system inputs cause only small changes in its outputs. Robustness is key to designing systems that work under uncertain or imprecise environments. While continuous control design algorithms can explicitly incorporate robustness as a design goal, it is not clear if robustness is maintained at the software implementation level of the controller: two "close" inputs can execute very different code paths which may potentially produce vastly different outputs.**

We present an algorithm and a tool to characterize the robustness of a control software implementation. Our algorithm is based on symbolic execution and non-linear optimization, and computes the maximum difference in program outputs over all program paths when a program input is perturbed. As a by-product, our algorithm generates a set of test vectors which demonstrate the worst-case deviations in outputs for small deviations in inputs. We have implemented our approach on top of the Splat test generation tool and we describe an evaluation of our implementation on two examples of automotive control code.

I. Introduction

A *cyber-physical system* (CPS) is an application that tightly co-ordinates discrete computation and continuous control of physical resources. Applications of CPS are numerous, ranging from autonomous vehicles and transportation networks to healthcare and assisted living.

One difficulty in implementing CPS is ensuring *robustness*. Robustness is the property ensuring that slight perturbations in the inputs to the system, e.g., noise in sensor measurements or actuator actions, cause only slight changes in the system execution. Robust designs are thus guaranteed to withstand uncertainty in the environment either by guaranteeing correct behavior or by guaranteeing that the resulting behavior only deviates modestly from the desired behavior upon the influence of small perturbations.

Consider the way complex control systems are designed and implemented. Usually, control design starts with the design of a mathematical model of the physical system (or *plant*) to be controlled, typically as differential equations whose solution gives the time evolution of the plant influenced by external inputs. The control engineer designs a *controller* that manipulates the external inputs to enforce a desired evolution of the plant. The controller can explicitly incorporate uncertainties in the system, arising either out of inadequate modeling of the plant (e.g., uncertain or complex dynamics that is not modeled, parameters known approximately due to measurement imprecisions). This is the field of *robust control*, which designs control laws that guarantee that the closed-loop behavior satisfies the objectives even in the presence of uncertainties. At this point, the field of robust feedback control is mature enough to answer many questions in robust control law design, based on efficient optimization algorithms and software tools [1], [2]. Once the control system has been designed using these tools, the system engineer implements the control law in software, by discretizing and sampling the physical signals, and by performing control computations in the software.

Unfortunately, the principles of robust design are not yet understood for *software implementations* of robust controllers. For software, there is no existing test for robustness or continuity, as input values that are physically very close can traverse entirely different code paths and can potentially produce very different outputs.

As an example of how interactions between the discrete and continuous worlds can cause loss of robustness, consider a luggage distribution system regulating the timing of interacting conveyor belts. For simplify, we assume that decisions to move a belt depends on the amount of time a shipment is waiting on the belt. In order to determine if there is a waiting shipment, belts are equipped with weight sensors. When the sensors report a weight value w above a certain threshold T, a shipment is considered to be present on the belt. Conversely, if $w \leq T$ it is assumed that no luggage is present. (In other words, the continuous state space is partitioned by the predicate $w \leq T$ for symbolic control.) Unfortunately, this sharp cutoff can cause two shipments with very similar weights to have very different behaviors as they progress through the system: a shipment of weight $T + \epsilon$ will have preference over a shipment of weight $T - \epsilon$ for any arbitrarily small $\epsilon > 0$. Although simplistic, the above example illustrates the difficulties arising from the interaction between physical and cyber systems. Close physical quantities are transformed into discrete decisions that lead to very different temporal behaviors.

Such examples indicate that software implementations should be checked for robustness, by varying inputs systematically and observing the relative differences in outputs. Unfortunately, the space of inputs is usually too big to be tested exhaustively. The problem is made difficult by many different and possibly correlated code execution paths, and control computations that depend on table lookups. This makes random testing ineffective in finding corner cases that break robust behavior.

In this paper, we study the *robustness verification* problem for control software. Let f be a function under test with inputs $x_1, \dots, x_n$ and output y. We say the function f is (δ, ϵ)-*robust* in the ith input if a difference of at most δ in the ith input (while all other inputs are identical) can cause a difference of at most ϵ in the output.

We give a test generation algorithm to compute the robustness ϵ of a function f given a perturbation δ. The input to our algorithm is a function under test f with inputs $x_1, \dots, x_n$ and output y, both inputs and outputs coming from an underlying metric space, and a tolerance δ. For each input x_i, our algorithm generates test vectors $(v_1, \dots, v_n)$ and $(v'_1, \dots, v'_n)$ that differ by at most δ in the ith 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.

Our algorithm is based on *symbolic execution* [3]–[6], which executes the program on symbolic inputs and aggregates symbolic constraints on each execution path. Unlike normal test generation, in which symbolic execution on individual paths is used to generate inputs that traverse the path, our algorithm looks at *pairs* of program paths. For each pair of program paths, we consider the symbolic constraints generated by symbolic execution, and maximize the difference in outputs on the two paths, subject to the symbolic constraints along the two paths and the following two additional constraints: (a) the ith input differs by at most δ in the two executions, and (b) all other inputs are identical along the two executions. The robustness of the program for tolerance δ is computed as the maximum over all pairs of paths of the maximum difference in output values.

We have implemented our algorithm on top of the Splat concolic test generation tool [7] and the Lindo non-linear optimization tool [8], and evaluated our algorithm on two examples of automotive control code: a fuel-air ratio controller and a transmission controller.

Our test generation technique and the computed robustness value can be used by control engineers in two ways. When the output depends robustly on an input in the mathematical model, our computation gives an indication of the "jitter" introduced by the implementation. For example, the desired clutch pressure in a transmission control program should depend continuously on the throttle angle, and our test set will show extremal situations for which the clutch pressure difference is maximal for small changes of the throttle angle. When an input changes control mode, the control engineer expects a discrete (but possibly bounded) change, and again, our test set and robustness value can indicate if this expectation is met. For example, in fuel-air ratio control, the (discrete) input can cause discrete but bounded changes in the output fuel level, and our robustness calculation again provides bounds on the possible change.

Robustness of hybrid automaton models for control systems have been studied before [9], [10], as have test generation for control software based on symbolic techniques [11], [12]. To the best of our knowledge, the problem of checking robustness in control software and generating tests that exhibit maximal output deviations has not been studied before.

II. Example

We illustrate our robustness analysis technique with a simplified version of a transmission shift control example from [13] (the full example is analyzed in Section V).

Figure 1 shows a transmission calculation example. The main function under test is *calc_trans_slow_torques*. It takes two inputs (throttle) *angle* and (vehicle) *speed*, and generates two outputs *pressure1* and *pressure2* (clutch pressures). The control computation uses precomputed *lookup tables*. A lookup table is a two-dimensional array in which the first column corresponds to inputs and the second column corresponds to outputs. There are three tables in the example: *data_table*, *out1_table* and *out2_table*. The lookup tables are accessed through the lookup functions *lookup1_2d* and *lookup2_2d*. The inputs to these functions are the address of a look-up table and an input value v. The lookup function finds out the element in the first column of the lookup table which is nearest to the input value v, and returns the corresponding value in the second column. The input *angle* is used to look up values in the table *data_table*. This is performed by the *lookup1_2d* function which divides the input domain into eight partitions, and returns the output value of the table for the partition to which the input belongs. For example, if the value of *angle* is 30, then the value of the variable *val1* becomes 41.

The value of *gear* is computed to be either 3 or 4 based on the relation between $3 \cdot$ *speed* and *val1*. This value is used to look up two other tables, *out1_table* and *out2_table*, one for setting the value of *val3* and the other for setting the value of *val4*. The output of looking up both tables *out1_table* and *out2_table* are either 0 or 1. Depending on this output, the output variables *pressure1* and *pressure2* can be either 0 or 1000. There are 64 execution paths in this code.

We want to analyze the effect of deviations in the inputs $angle$ and $speed$ on the outputs $pressure1$ and $pressure2$.

Let us first consider the output *pressure1*. Intuitively, for every value of the inputs, the variable *gear* can take any of the two values 3 and 4, but the lookup of table *out1_table*

```
OUTPUT  int pressure1, pressure2;
int data_table[8][2] =
{ {10, 29}, {25, 29}, {30, 41}, {40, 63},
{60, 109}, {70, 127}, {80, 127}, {100, 127} };

int out1_table[4][2] =
{ {1, 0}, {2, 0}, {3, 1}, {4, 1} };

int out2_table[4][2] =
{ {1, 0}, {2, 0}, {3, 0}, {4, 1} };

int lookup1_2d(int *table_address, int xval)
{
  int yval, index;
  if (xval < 17) index = 1;
  else if ((xval >= 17) && (xval < 27)) index = 3;
  else if ((xval >= 27) && (xval < 35)) index = 5;
  else if ((xval >= 35) && (xval < 50)) index = 7;
  else if ((xval >= 50) && (xval < 65)) index = 9;
  else if ((xval >= 65) && (xval < 75)) index = 11;
  else if ((xval >= 75) && (xval < 90)) index = 13;
  else index = 15;
  yval = *(table_address + index);
  return yval;
}

int lookup2_2d(int *table_address, int xval)
{
  int yval, index;
  if (xval == 1) index = 1;
  else if (xval == 2) index = 3;
  else if (xval == 3) index = 5;
  else if (xval == 4) index = 7;
  yval = *(table_address + index);
  return yval;
}


int calc_trans_slow_torques(int angle, int speed)
{
  int val1, val3, val4;
  int gear;
  val1 = lookup1_2d(&(data_table[0][0]), angle);
  if ( 3 * speed <= val1 )
     gear = 3;
  else
     gear = 4;
  val3 = lookup2_2d(&(out1_table[0][0]), gear);
  pressure1 = val3 * 1000;
  val4 = lookup2_2d(&(out2_table[0][0]), gear);
  pressure2 = val4 * 1000;
}
```

Figure 1. An example to illustrate robustness analysis of a program

with either value results in the same output, and hence the value of *pressure1* is identical. So, we conclude that the output *pressure1* is robust to perturbations of its inputs.

Let us now analyze the effect of input deviations on the output *pressure2*. When the actual values of *angle* and *speed* are 34 and 14 respectively, the value of *gear* is set to 4. So, the lookup of table *out2_table* with $gear = 4$ returns 1, and the value of the output variable *pressure2* is 1000. Now suppose that the input variable *angle* is measured wrongly as 35 instead of 34. In this case, *lookup1_2d* function returns 63 instead of 41. In this case, the value of *gear* is set to 3. Now, lookup of table *out2_table* with $gear = 3$ yields 0, which makes the value of the output variable *pressure2* equal to 0. That is, a deviation of one unit in the angle can cause a deviation of 1000 units in the output $pressure2$.

Similarly, suppose that the input *angle* is measured correctly, but the input *speed* is wrongly measured to be 13 instead of 14. So, the value of *val2* becomes 39, whereas the value of *val1* remains 41. As a result, the value of *gear* is again set to 3 instead of 4, and the value of the output *pressure2* becomes 0 instead of 1000 as before. This indicates that a small deviation in *speed* can again cause a large deviation in the output *pressure2*.

The example demonstrates that the sensitivity of program outputs to program inputs can be non-obvious. but checking sensitivities can be complicated due to (a) many (inter-procedural) code paths in the program and (b) data flow in the program through table lookups. Our algorithm symbolically traverses program paths and collects constraints on input and output variables. For each *pair* of program paths, our algorithm determines values of input variables that cause the program to follow these two paths and for which the difference in values of the output variable is maximized.

For example, consider the following two (simplified) symbolic constraints arising from two possible executions computing $pressure2$:

$$\begin{aligned} & angle \geq 17 \wedge angle \geq 27 \wedge angle < 35 \wedge val1 = 41 \wedge \\ & 3 \cdot speed > val1 \wedge gear = 4 \wedge val4 = 1 \wedge \\ & pressure2 = val4 \cdot 1000 \end{aligned} \tag{1}$$

and

$$\begin{aligned} & angle' \geq 17 \wedge angle' \geq 27 \wedge angle' \geq 35 \wedge angle' < 50 \wedge \\ & val1' = 63 \wedge \\ & 3 \cdot speed' \leq val1' \wedge gear' = 3 \wedge val4' = 0 \wedge \\ & pressure2' = val4' \cdot 1000 \end{aligned} \tag{2}$$

corresponding to the two execution paths described above for $angle = 34$ and $angle = 35$ respectively. (The variables in the second constraint have been renamed to "primed" versions.) In order to check input sensitivity, we formulate the optimization problem in which we maximize $|pressure2 - pressure2'|$, subject to the constraints (1) and (2), and additionally the constraints

$$speed = speed'$$

stating that the input *speed* does not change and

$$|angle - angle'| \leq 1$$

stating that the deviation in $angle$ is at most 1 unit. The solution of the optimization problem gives $angle = 34$, $angle' = 35$, and $speed = speed' = 14$, for which the outputs differ by 1000 units. We iterate over all such path pairs and calculate the maximum deviation in outputs for a deviation of 1 unit in the inputs. We find in this example that the deviation in $pressure2$ is 1000 units.

A similar analysis with $pressure1$ shows that the deviation is 0, as expected by our informal reasoning.

III. Preliminary Definitions

A. Program Representation

We illustrate our algorithm on a simple imperative language. In our implementation we handle more general features such as pointers, arrays, and function calls.

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

1) an assignment $x := e$, where $x \in X$ and e is an arithmetic expression over X,
2) a conditional $\mathsf{if}(e)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$, where e is a side-effect free expression and ℓ', ℓ'' are locations in $\mathcal{L}$,
3) an output statement $\mathsf{output}(y)$, where $y \in X$ is 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 ℓ where $\mathsf{op}(\ell)$ is an assignment statement has exactly one node ℓ' with $(\ell, \ell') \in E$, (ii) every node ℓ such that $\mathsf{op}(\ell)$ is $\mathsf{if}(e)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$ has two edges (ℓ, ℓ') and (ℓ, ℓ'') in E. For a location $\ell \in \mathcal{L}$ where $\mathsf{op}(\ell)$ is an assignment operation, we write $N(\ell)$ for its unique neighbor. We assume in the following that ℓ_o is the only location labeled with the output statement.

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. The program ends on reaching the location ℓ_o and outputs the value of y. A *path* is a sequence of locations $\ell^1, \ell^2 \ldots \ell^n$ in the CFG. A location $\ell \in \mathcal{L}$ is reachable from $\ell' \in \mathcal{L}$ if there is a path $\ell' \ldots \ell$ in the CFG. We assume that every node in $\mathcal{L}$ is reachable from ℓ_i and ℓ_o is reachable from every node. In what follows, we assume that every program execution terminates.

Semantics. 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 $M(z)$. For an expression e, we denote by $M(e)$ the value obtained by evaluating e where each variable x occurring in e is replaced by the value $M(x)$.

Execution starts from a memory M_0 containing initial values for input variables in X_0 and constant default values for variables in $X \setminus X_0$, at the entry location ℓ_i. Each operation updates the memory and the control location. Suppose the current location is ℓ and the current memory is M. If $\mathsf{op}(\ell)$ is $x := e$, then the new location is $N(\ell)$ and the new memory is $M[x \mapsto M(e)]$. If $\mathsf{op}(\ell)$ is $\mathsf{if}(e)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$ then e is evaluated based on the current memory M. If the evaluated value is 0, then the new location is ℓ'', otherwise the new location is ℓ'. In either case, the memory remains unchanged. If $\mathsf{op}(\ell)$ is the output statement $\mathsf{output}(y)$, 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

Our algorithm for robustness analysis is based on *concolic execution*, in which the program is executed on symbolic inputs in addition to concrete inputs [4]–[6]. We briefly recall the basic algorithm, and use it as a subroutine later.

The concolic execution algorithm executed the program while maintaining two additional artifacts: a *symbolic memory* μ which maps variables in X to symbolic expressions over a set of symbolic constants, and a *path constraint* ξ, 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 ℓ_i with the initial memory M_0. Initially, the symbolic memory μ maps each input variable $x \in X_0$ to a fresh symbolic constant α_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 μ is updated to $\mu[x \mapsto \mu(e)]$, where $\mu(e)$ denotes the symbolic expression obtained by evaluating e using the map μ. The path constraint is unchanged. The control location is updated to $N(\ell)$.

For a conditional $\mathsf{if}(e)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$, there is a choice in updating the control location. If the concrete execution chooses the new control location to be ℓ' (i.e., if $M(e) \neq 0$), the path constraint is updated to $\xi \wedge \mu(e) \neq 0$, and if the new control location is ℓ'' (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 μ.

Execution ends when the control location is ℓ_o. At this point, the symbolic output is $\mu(y)$ and the concolic execution routine returns the pair $\langle \mu(y), \xi \rangle$ of the symbolic output and the path constraint.

At the end of an execution, a new execution is created by selecting a conditional $\ell : \mathsf{if}(e)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$ 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 ℓ but then takes the "else" (respectively, "then") branch of this conditional has not been explored before. Let ξ_ℓ be the symbolic constraint just before executing this conditional and ξ_e be the constraint generated by the execution of this conditional. We find a satisfying assignment for the constraint $\xi_\ell \wedge \neg\xi_e$. 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 location ℓ, 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.

At the end of concolic execution, we get a set of pairs $\langle e, \xi \rangle$ of symbolic output expressions and path constraints for each explored path.

IV. Robustness Analysis

A. Problem Definition

In this section we formally define the problem of robustness analysis. Let $P = (X, X_0, y, \mathcal{L}, \ell_i, \ell_o, \mathsf{op}, E)$ be a program. Let us fix an input $x \in X_0$. Let δ_x denote the maximum possible uncertainty in measuring input x. Measurement errors in x can cause the output y 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 y is data-dependent on the value of x. We define the *maximum output sensitivity* of y w.r.t. x and δ_x as follows:

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

where x and x' denote the actual and measured value of x respectively, v is the value of all other input variables, and y and y' denote the value of the output of the program for input v, x and v, x' respectively. Informally, δ_{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 δ_x. A program P is (δ, ϵ)-*robust* with respect to an input $x \in X_0$ if $\delta_{yx} \leq \epsilon$ whenever $\delta_x \leq \delta$.

B. Algorithm

The algorithm for finding maximum sensitivity of output y of a program with respect to an input x is presented in Algorithm IV.1. First, the algorithm performs concolic execution on the program and collects a set S of pairs $\langle e, \xi \rangle$ of the output symbolic expression e and the path constraint for each path (line 2).

For each pair $\langle e_1, \xi_1 \rangle$ and $\langle e_2, \xi_2 \rangle$ from S, the algorithm finds out the maximum deviation in output y when the input x is bounded by δ_x and the two execution paths are assumed to satisfy ξ_1 and ξ_2. This is done by procedure *find_output_deviation* (line 6) described in the next section. Finally, the maximum output sensitivity for y w.r.t. input x is defined as the maximum over all pairs of paths in S of the deviations. The computation of δ_{yx} is interleaved in lines 3 and 7.

```
1  Algorithm find_output_sensitivity(P, x, δ_x)
2  S = concolic(P);
3  δ_yx = 0;
4  for ⟨e1, ξ1⟩ ∈ S do
5      for ⟨e2, ξ2⟩ ∈ S do
6          Δ =
           find_output_deviation(⟨e1, ξ1⟩, ⟨e2, ξ2⟩, δ_x);
7          δ_yx = max(δ_yx, Δ);
8      end
9  end
10 return δ_yx
```

Algorithm IV.1: Algorithm to find out maximum deviation in output y for perturbation δ_x in input x in a Program P

C. Finding Maximum Output Deviation

We now describe how the *find_output_deviation* function in Algorithm IV.1 works. The inputs of this function are two pairs $\langle e_1, \xi_1 \rangle$ and $\langle e_2, \xi_2 \rangle$ from S, and the deviation δ_x in the value of x. The function finds the maximum output deviation when deviation in input x is bounded by δ_x.This is done by formulating and solving several maximization problems.

For a symbolic expression e, let the *primed version* e' denote the symbolic expression in which every variable α in e is replaced by a primed copy α'. Similarly, for a path constraint C, let the *primed version* C' denote the same constraint in which every variable α is replaced by a primed copy α'. For example, the primed version of $x + y$ is $x' + y'$ and the primed version of $x \geq 0 \wedge y = x$ is $x' \geq 0 \wedge y' = x'$.

We now define the maximization problem. The objective function is $|e_i - e'_j|$, which represents the difference between the symbolic outputs along the two paths (note that e'_j is primed).

The constraints of the optimization problem are

$$\xi_1 \wedge \xi'_2 \wedge \bigwedge_{z \in X_0 \setminus \{x\}} \alpha_z = \alpha'_z \wedge |\alpha_x - \alpha'_x| \leq \delta_x \qquad (3)$$

The first two conjuncts enforce that the two execution paths follow the path constraints ξ_1 and ξ_2 respectively, the third conjunct $\bigwedge_{z \in X \setminus \{x\}} \alpha_z = \alpha'_z$ 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 δ_x.

Thus, the optimization problem is:

$$\begin{array}{ll} maximize & |e_1 - e'_2| \\ s.t. & \text{constraints (3)} \end{array}$$

The optimization problem above may not always be satisfiable. For example, if ξ_1 contains a constraint $\alpha_z < 0$ and ξ_2 contains a constraint $\alpha_z \geq 0$, where $z \in X$ and z is independent from the variable x for which we are measuring output sensitivity, then constraints (3) is not satisfiable. If the constraints (3) are unsatisfiable, we assume that the function $find\_output\_deviation$ returns -1, otherwise it returns the solution of the optimization problem. Note that just because two paths diverge at a conditional does not mean that the constraints are unsatisfiable for these paths. Suppose that two paths diverge at a conditional $\alpha_x < 0$, i.e., the path constraint ξ_1 has the constraint $\alpha_x < 0$ and the path constraint ξ_2 the constraint $\alpha_x \geq 0$. In the optimization problem, $\alpha_x \geq 0$ is replaced by $\alpha'_x \geq 0$, and we have an additional constraint of the form $\alpha_x - \alpha'_x \leq \delta_x$. Thus, the constraints can be satisfiable in this case for inputs α_x and α_x that are "close" (within δ_x) and follow the two program execution paths.

V. Implementation and Case Studies

A. Implementation

We have implemented a tool to analyze the robustness of C programs based on the algorithm described in Section IV. We extend Splat [7] to generate the symbolic output expression and the path constraints for all program paths. Splat uses CIL [14] for internal representation of a C program. Splat performs concolic execution by running the program on concrete as well as symbolic inputs. The decision procedure STP [15] is used to solve the path constraints to generate inputs for a new execution. After traversing a path, when the Splat run exits, we store the output symbolic expression and the path constraint. In addition to the path constraint, we add additional restrictions on the range of inputs when they are known from the problem domain.

We use the Lindo optimization API [8], which provides general nonlinear and nonlinear/integer optimization problem solving APIs, for solving the maximization problems. Lindo can detect if a set of constraints is not satisfiable. If the set of constraints is not satisfiable, Lindo reports that there is no satisfying assignment. If the non-linear model is feasible, Lindo provides the solution of the maximization problem, with the corresponding values for the variables in the optimization problem.

Lindo does not accept strict relatios "$<$", "$>$", and "$\neq$" in the constraints. So, in our implementation, we convert the constraints containing these relational operators to suitable constraints containing only "$\leq$" and "$\geq$" by introducing a small valued constant α. If we have a constraint of the form "$g < b$" then we convert it to "$g + \alpha \leq b$". Similarly, a constraint of the form "$g > b$" is converted to "$g - \alpha \geq b$". Finally, if a set of constraints contains a constraint of the form "$g \neq b$", we form two maximization problems with the same objective function. In one problem, we keep "$g + \alpha \leq b$" instead of "$g \neq b$" in the set of constraints, and in the other problem we keep "$g - \alpha \leq b$" instead of "$g \neq b$" in the set of constraints.

Normally, Splat performs concolic execution for path coverage. We extend the test generation algorithm to additionally generate test cases that hit all look-up table entries. Since the size of each look-up table is known statically, the extension involves adding a constant number of disjunctive constraints at each table look-up that ensures that different tests hitting different table entries are generated.

B. Case Studies

We carry out two case studies to judge the applicability and scalability of robustness analysis to real-world control programs. We perform these case studies on two C programs present in Ford's "Smart Vehicle Baseline Report" [13]. The code for both examples was generated automatically by TargetLink [16] from two Powertrain control models in Simulink: (i) Fuel-Air Ratio Control and (ii) Transmission Shift Control.

Fuel-Air Ratio Control. The Fuel-Air Ratio Control model is primarily a data flow model, i.e., the output is derived from inputs almost exclusively by the evaluation of algebraic equations. As a component of the overall controller, the function of this model is to calculate desired fuel (*dMfc*) in kg/second, by taking the following as inputs to drive the calculation: *we*: engine speed (in radians/sec), *map*: manifold absolute pressure (in kilopascals), *bp*: barometric pressure (in kilopascals), *tps*: throttle position sensor (in degrees), *o2s*: oxygen threshold sensor (logical/Boolean). However, an analysis of the C code generated for the model reveals that only the first two inputs influence the calculation of *dMfc*.

In [13], both floating point and fixed-point code generated by TargetLink are available. We choose the fixed-point version, as Splat does not support floating point variables and constants due to limitations of the underlying decision procedure Stp. In the fixed-point C code generated by TargetLink, the floating point variables are represented as 16-bit signed integers. The most significant bit of the integer represents the sign bit. The position of the virtual binary point in a variable is decided by the range of the absolute value of the integer part of the variable. For example, if the absolute value of the integer part of a floating point variable is bounded by 127, then the position of the virtual binary point is after 8-th MSB (the MSB is used for the sign bit, and the next 7 bits are used to capture the integer part). The fixed-point representation of a floating point variable thus can uniquely be specified by the weight on the LSB of the fixed-point integer. For example, a fixed-point integer for which the weight on the LSB is 2^{-7} represents a floating point variable which can take value in the range of -256 to 255.9921875. During an arithmetic operation the position of the binary point is maintained by logical shift operations. The inputs to the program, *we* and *map*, are both positive quantities, but with different range and precision. Their minimum value is 0 and maximum value is the largest sixteen bit positive integer. We specify these lower and upper bounds in the maximization problems.

The Fuel-Air Ratio Control model implements a feedback loop, so the values of state variables in one invocation are read and used in the next invocation. If we model the infinite loop, then path generation by concolic execution does not terminate (there are an infinite number of paths). Hence, in our experiments, we unroll the loop two iterations.

Splat gives us three unique paths in the program. Our manual analysis revealed that the program actually has five unique feasible paths. The two missing paths are due to limitations of the underlying decision procedure Stp in dealing with complex nonlinear algebraic conditions. The output expressions and the constraints for the three paths obtained from Splat are used to form six maximization problems for the six path pairs, denoted pp_{ij} for $i, j \in \{1, 2, 3\}$ and $i \leq j$. Lindo detects that among these six path pairs, four path pairs pp_{ij} for $i \neq j$, and pp_{22} are not feasible.

Notice that the infeasibility of the path pair pp_{22} should indicate the infeasibility of path p_2, and Splat should have not found such a path. It turns out that in the path constraint for p_2 has a constraint $\alpha \neq 0$, where the symbolic variable α denotes the value of a variable which only takes non-negative values. Splat finds a feasible execution of path p_2 when the variable α is set to a negative value. However, in the maximization problem, we explicitly add the constraint that $\alpha \geq 0$, and this causes the path pair pp_{22} to become infeasible.

Among the two feasible path pairs pp_{11} and pp_{33}, the maximum output sensitivity for pp_{11} is 0 for both the inputs *we* and *map*. The set of symbolic path constraints for path pair pp_{11} contains a constraint $sym_1 = 0$, where sym_1 denotes the symbolic value for input *we*. As the input *we* is zero for path p_1, due to deviation in this input, the output does not change along this path. Moreover, when the input *we* is 0, the output of the program along path p_1 is evaluated to a constant value. That is why, for the deviation in the second input, there is no deviation in the output.

In case of path pair pp_{33}, the maximum output deviation for input *we* is 4, and for input *map* is 77, when the input deviation is bounded by 1 in both the cases. In the fixed point representation of *we* and *map*, the weights on the LSB are 2^{-4} and 2^{-7} respectively. So, deviations of 1 unit in the fixed point representations of these inputs are equivalent to 2^{-4} and 2^{-7} unit of deviation in the actual values. The weights on the LSB in the fixed point representation of output *dMfc* is 2^{-20}. So, the deviation of 4 unit and 77 unit in the fixed point representation of the output is equivalent to the deviations of $3.81e - 6$ and $7.34e - 5$ respectively. Moreover, by varying the amount of deviation, we find that the maximum output deviation grows linearly with the deviation in both the inputs.

Transmission Shift Control. Our second case study is the automatically generated floating point C code from an automatic transmission model. Though the generated code uses floating point variables and constants, they either denote integer or boolean quantities, or can be approximated to integer values without changing the functionality required for our analysis. The model describes controls for a 4-speed automatic transmission system. The system has the following four inputs: *throttle angle* (in degrees), *vehicle speed* (meter per second), *state of a power/economy switch* (logical), and *actual gear commanded* (1, 2, 3, or 4). Among these four inputs the first two inputs come from sensor readings. Hence we analyze the sensitivity of the outputs of the system on these two inputs. The system has five outputs: these are five different clutch pressures. They are denoted as $pb12$, $pc1$, $pc2$, $pc3$, and $pc4$. These five outputs are independent of each other, so we analyze the sensitivity of one of them at a time.

The control program selects a gear based on the input values and computations based on table lookups. The selected gear is used to look up a Boolean value for each of the output clutch pressure. The Boolean value decides if the output clutch pressure is 0 or 1000. Depending on the current condition of the gear, the system may be in one of 10 states: it is either on one of the four gears, or transitioning from one gear to another (six possibilities). We analyze each of these ten cases separately. We first use Splat to generate output values and path constraints for all the paths. Then we separate the path information for different current gear condition before further analysis.

In the maximization problems, we bound the value of *throttle angle* to between 0 and 120 degrees, and the value of *vehicle speed* between 0 to 80 m/sec. The value of *throttle angle* is used as the input to some look-up tables containing calibration parameters used in the procedure for deciding the next gear. The first column of each look-up table corresponds to inputs and the second column corresponds to calibration parameters. Each look-up table has eight rows corresponding to eight discrete values for *throttle angle* (the minimum one is 10 and the maximum one is 100). Let us denote the look-up table entries by $(t_1, v_1), \ldots (t_8, v_8)$, where t_i denotes a value of *throttle angle*, and v_i denotes the corresponding calibration parameter. The value of *throttle angle* is used as input values for the table look-up. It is compared to the values in the first column of the look-up table. If the value is nearest to t_i in the first column of the look-up table, then v_i is returned as the output of the look-up. So, the deviation in the measurement of *throttle angle* may lead to fetching of wrong data from the look-up tables, leading to selection of wrong gear. The value of *vehicle speed* is used to compute an expression that is used to compare with the fetched value from the look-up table. So, deviation in the measurement of *vehicle speed* may also lead to execution of wrong path in the program.

The outputs of the program are either 0 or 1000. So along the same path, input deviation does not cause any output deviation. Output deviation happens only if a different path is executed because of input perturbations. We are interested to check if the program is $(\_, 0)$-robust. We write "_" to denote any non-zero value for input deviation. An output

Current Gear Condition	*# Paths*	*# Path Pairs*	*# Infeasible Path Pairs*	*# Insensitive Path Pairs*	*# Sensitive Path Pairs*
1	16	136	105	31	0
2	24	300	249	51	0
3	24	300	247	45	8
4	16	136	102	30	4
$1-2$	64	2080	1878	202	0
$2-3$	64	2080	1842	208	30
$3-4$	64	2080	1851	190	39
$4-3$	64	2080	1866	181	33
$3-2$	64	2080	1852	193	35
$2-1$	64	2080	1704	376	0

Table I
ROBUSTNESS ANALYSIS FOR TRANSMISSION MODEL FOR INPUT *throttle angle* AND OUTPUT *pb12*

Current Gear Condition	*# Paths*	*# Path Pairs*	*# Infeasible Path Pairs*	*# Insensitive Path Pairs*	*# Sensitive Path Pairs*
1	16	136	112	24	0
2	24	300	252	48	0
3	24	300	252	32	16
4	16	136	122	16	8
$1-2$	64	2080	1920	160	0
$2-3$	64	2080	1920	128	32
$3-4$	64	2080	1920	104	56
$4-3$	64	2080	1936	88	56
$3-2$	64	2080	1936	88	56
$2-1$	64	2080	1816	264	0

Table II
ROBUSTNESS ANALYSIS FOR TRANSMISSION MODEL FOR INPUT *vehicle speed* AND OUTPUT *pb12*

of the program is not robust for an input if the program is $(\_, 1000)$-robust for the input-output pair.

Table I and Table II shows the number of infeasible, insensitive and sensitive path pairs in the program for input *throttle angle* and *vehicle speed* respectively, and for output *pb12*. In both the tables, the first column shows the current gear condition, the second column shows the number of unique paths for the current gear condition, and the third column shows the number of possible path pairs for the same current gear condition. The fourth, fifth and the sixth column show the number of infeasible, insensible and sensible path pairs for the output with respect to the chosen input. By insensitive path pair, we mean that though it is possible that one path in the pair is executed instead of the other path due to small drift in input, the output value does not change. By sensitive path pair, we mean that it is possible that the program executes along the wrong path in the pair due to small deviation in input measurement, and also the output deviates from the actual output. Clutch pressure *pb12* is used when the gear is in first or second position. From Table I and table II, we get that when any of the first two gears is involved in the current gear condition, the number of sensitive path pairs is 0 for both *throttle angle* and *vehicle speed*. This ensures that the system is $(\_, 0)$-robust for output *pb12*.

Table III shows the number of sensitive path pairs for other clutch pressure outputs in different gear conditions for both *throttle angle* and *vehicle speed*. Clutch pressure *pc1* is used for gear 1 and 2, *pc2* is used for gear 2, 3 and 4, *pc3* is used for gear 3 and 4, and *pc4* is used for gear 4. From the number of sensitive path pairs for different output clutch pressures in different gear conditions, it is evident that the system is also $(\_, 0)$-robust for output *pc1* for both the inputs, but the system is $(\_, 1000)$-robust for the other output clutch pressures for both the inputs.

Input	$throttle\ angle$				$vehicle\ speed$			
Gear	$ClutchPressures$							
Cond	$pc1$	$pc2$	$pc3$	$pc4$	$pc1$	$pc2$	$pc3$	$pc4$
1	0	0	0	0	0	0	0	0
2	0	6	0	0	0	16	0	0
3	8	8	8	0	16	16	16	0
4	4	4	4	4	8	8	8	8
$1-2$	0	18	0	0	0	32	0	0
$2-3$	30	45	30	0	32	56	32	0
$3-4$	39	39	39	27	56	56	56	32
$4-3$	33	33	33	21	56	56	56	56
$3-2$	35	45	35	0	56	56	56	0
$2-1$	0	13	0	0	0	104	0	0

Table III
NUMBER OF SENSITIVE PATH PAIRS FOR *pc1*, *pc2*, *pc3*, *pc4* IN DIFFERENT GEAR CONDITIONS

VI. CONCLUSION AND LIMITATIONS

In this paper, we have taken a preliminary step toward analyzing software programs for robustness. While we describe our algorithm using a concrete distance function, our robustness analysis algorithm can be extended for inputs and outputs coming from general metric spaces by setting up the optimization problem *maximize* $L_o(y, y')$ subject to $\xi_1 \wedge \xi_2 \wedge L_i(x, x') \leq \delta$, where L_i and L_o denote metrics on the inputs and outputs respectively.

At this point, our implementation has several limitations. First, our underlying decision procedure handles fixed point but not floating point numbers and non-linear arithmetic. Second, we treat δ_x as a constant, and compute the sensitivity relative to this constant. The behavior of the output sensitivity as the input deviation varies can be plotted by running our analysis for a range of δ_x values. Ideally, we would like to get a function giving δ_{yx} as a symbolic function of δ_x. In many cases, this is possible using *parametric* non-linear optimization methods [17], but we have not explored this direction.

Third, we do not have a satisfactory way to analyze closed loop systems. In our robustness analysis, we consider input state variables as independent inputs and output state variables as independent outputs, and compute robustness for one iteration of the control computation. Within a control function, loops usually have constant bounds and are unrolled by concolic execution. However, for a closed loop control system, where outputs of one iteration may be stored in state variables and used in computations in the next iteration, we cannot argue directly that two "close" inputs at the beginning of iteration 0 remain close for every iteration. For example, consider a control computation for which a deviation of δ in the input causes a deviation of 2δ in the output. If the input at iteration $i+1$ is the output from iteration i, then in n iterations of the computation, two initial inputs δ-apart will be $2^n\delta$ apart. In the case study, we perform a "bounded" analysis by unrolling the control iterations for a constant number of cycles, which does not guarantee closeness for all iterations, but provides some intuition into the program behavior. An alternative approach would try to prove a stronger property for the function, e.g., that it is non-expanding.

Acknowledgments. This research was funded in part by the NSF awards CNS-0720881 and CCF-0702743.

REFERENCES

[1] S. Boyd, L. E. Ghaoui, E. Feron, and V. Balakrishnan, *Linear Matrix Inequalities in System and Control Theory*, ser. Series in Applied Mathematics, vol 15. SIAM, 1994.

[2] C.-Y. Kao, A. Megretzki, U.Jönsson, and A. Rantzer, "A Matlab toolbox for robustness analysis," in *IEEE International Symposium on Computer-Aided Control Systems Design*. IEEE, 2004.

[3] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19(7), pp. 385–394, 1976.

[4] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing." in *PLDI*, 2005.

[5] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *FSE*, 2005.

[6] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "Exe: automatically generating inputs of death." in *CCS*, 2006.

[7] R. Xu, P. Godefroid, and R. Majumdar, "Testing for buffer overflows with length abstraction," in *ISSTA*, 2008, pp. 27–38.

[8] LINDO Systems Inc., "LINDO API 6.0 - powerful library of optimization solvers and mathematical programming tools," http://www.lindo.com/.

[9] V. Gupta, T. Henzinger, and R. Jagadeesan, "Robust timed automata," in *Hybrid and Real-time Systems*, ser. Lecture Notes in Computer Science 1201. Springer, 1997, pp. 331–345.

[10] E. Frazzoli, M. Dahleh, and E. Feron, "Robust hybrid control for autonomous vehicle motion planning," in *IEEE Conference on Decision and Control vol. 1*. IEEE, 2000, pp. 821–826.

[11] S. Xia, B. D. Vito, and C. Muñoz, "Automated test generation for engineering applications," in *ASE 05: Automated Software Engineering*. ACM, 2005, pp. 283–286.

[12] S. Xia, B. D. Vito, and C. Muñoz, "Predicate abstraction of programs with non-linear computation," in *ATVA 06: Automated Technology for Verification and Analysis*, ser. Lecture Notes in Computer Science, vol. 4218. Springer-Verlag, 2006, pp. 352–368.

[13] D. Bostic, W. P. Milam, Y. Wang, and J. A. Cook, "Smart vehicle baseline report," 2001, http://vehicle.berkeley.edu/mobies.

[14] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *CC*, 2002.

[15] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV*, 2007.

[16] dSPACE, "TargetLink," http://www.dspaceinc.com/ww/zh/zho/home/products/sw/pcgs/targetli.cfm?nv=bbp.

[17] H. Jongen and G.-W. Weber, "On parametric nonlinear programming," *Annals of Operations Research*, vol. 27, pp. 253–284, 1990.