When does my program fail?

Jeremias Rößler
Department of Computer Science
Saarland University
Saarbrücken, Germany
roessler@cs.uni-saarland.de

Alessandro Orso
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
orso@cc.gatech.edu

Andreas Zeller
Department of Computer Science
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

***Abstract*—Oops! My program fails. Which are the circumstances under which this failure occurs? Answering this question is one of the first steps in debugging—and a crucial one, as it helps characterizing, understanding, and classifying the problem. In this paper, we propose a technique to *identify failure circumstances automatically*. Given a concrete failure, we first compute the *path condition* leading to the failure and then use a constraint solver to identify, from the constraints in the path condition, the general *failure conditions:* "The program fails whenever the credit card number begins with 6, 5, and a non-zero digit." A preliminary evaluation of the approach on real programs demonstrates its potential usefulness.**

***Index Terms*—automated debugging; symbolic execution; constraint solving;**

I. Introduction

When a program fails, one of the first steps in debugging the problem is to identify the exact circumstances under which the failure occurs. These circumstances not only help understanding the problem for the actual debugging work; they also are important to assess the severity of the problem and help in identifying related or duplicate problems.

In a recent study at Microsoft [1], knowing "in what situations a failure occurred" was the third most frequent unsatisfied information need of developers, and by far the most time-consuming of all. Why is this so difficult? As a simple real-life example, consider the `processCard` program, which takes as input a credit card number to be processed, checks whether the number is valid using the Luhn formula (a simple checksumming algorithm), and identifies the card type. Unfortunately, `processCard` contains a bug, whose effects are shown in Figure 1. The credit card number given as input is a valid Discover card number, but it is rejected by the application:

```
$ ./processCard '6510 2556 8418 3585'
InaccurateCardType: Discover 6510 2556 8418 3585
$ _
```

Fig. 1. `processCard` erroneously rejects a valid credit card number

At this point, we have everything in place to start our investigation. We have a failing application and a test input that allows us to reproduce the failure at will, but we do not know under which circumstances the program fails. We do not know whether the failure would occur for other credit card numbers and, if so, for which ones. It is also possible, although unlikely, that the failure may occur for this very specific number only, in which case we might focus on more prevalent issues first. Therefore, even before we start the actual debugging, *we need to understand the failure circumstances.*

In this paper, we present an approach to derive such failure circumstances automatically. Given a concrete failing run, we compute the *path condition* leading to the failure—a conjunct of constraints on the inputs under which the control flow reaches the location where the specific failure occurs with the right state. This conjunction of constraints provides the conditions for the failure to occur.

As this conjunction can become arbitrary complex, it is necessary to *simplify and generalize* the collected path conditions, which is a challenging endeavor. In our current approach, we perform this simplification and generalization using a combination of systematic experimentation, constraint solving, and constraint abstraction. (We are also experimenting with alternative techniques to gain further insights into the problem.) For relatively simple programs, our approach can already produce a short and crisp description of the failure circumstances. For the failure in Figure 1, for instance, the computed failure condition is

$$\mathit{isValidCardNumber}(v) \land v_0 = \texttt{'6'} \land v_1 = \texttt{'5'} \land v_2 \neq \texttt{'0'}$$

which means that the failure occurs whenever the card number (1) is valid according to the Luhn formula, (2) starts with `65`, and (3) has the third digit different from zero. Note that this is a realistic example. On October 2006, the prefix for Discover credit cards changed from "650" to "65". Because the `processCard` code was not updated to reflect this change, valid Discover card numbers that start with "65[1–9]", such as the number in our test input, are not correctly processed and result in an exception being thrown. The predicate that our approach computes accurately reflects this failure condition.

In the remainder of this paper, we first state the problem we target, introducing a set of basic definitions (Section II). We then illustrate our approach in more detail on the `processCard` example (Section III). In Section IV, we provide details on how to simplify constraints. Section V gives insights into an additional real bug and directly leads to the discussion of current challenges and future work (Section VI). Finally, Section VII discusses related work, and Section VIII presents conclusions and consequences.

II. Definitions and Terminology

Before illustrating our approach, we define the terminology that we use in the rest of the paper.

We define a *program*: $I \rightarrow O$ as a function from I (the program domain) to O (the program co-domain). An *oracle*: $I \rightarrow O$ for a program p is also a function from I to O that defines, for each $i \in I$, which $o \in O$ should be produced by p. Given a program p, I and O for p, and a corresponding oracle *orac*, we can define a *failure* as a pair $\langle i, o \rangle$ such that $i \in I$, $o \in O$, $p(i) = o$, and $orac(i) \neq o$.

A *constraint* is a predicate defined over a set of variables, where each variable is defined over a specific domain. In our specific context, the variables within the condition correspond to program inputs, and constraints correspond to predicates used in conditional statements (*e.g.,* the predicate of an `if`-statement) within the program.

A *path condition* (PC) is a conjunction of constraints, where the constraints correspond to predicates encountered while following a specific path. A PC, by defining a set of constraints on the inputs of a program, identifies a (possibly empty) subdomain of the program with the following property: each input in that subdomain causes the program to follow the path that corresponds to the PC.

A *solution* for a PC (*i.e.,* for the corresponding conjunction of constraints) is an assignment of a single value from its domain to each variable in the constraints such that no constraint is violated. In our context, a solution is therefore a set of values for the inputs of the program that satisfy all constraints in the PC, that is, that cause the program to follow the path corresponding to the PC. A PC may have one, many, or no solutions. In the first two cases, the PC is said to be *satisfiable* (or *consistent*). In the latter case (*i.e.,* if there is no possible assignment of values to variables that satisfies all the constraints), the PC is said to be *unsatisfiable* or *inconsistent*.

We now state our goal in more formal terms using the above definitions. Given a program p, an oracle *orac*, and an input i_f that results in a failure $f = \langle i_f, o_f \rangle$, we first want to find a set of constraints C_{min} with the following two properties:

1) $\forall i \in I$ that satisfies $C_{min} : p(i) \neq orac(i)$, $p(i) = o_f$
2) $\forall c \in C_{min} : C_{min} - \{c\}$ does not satisfy Property 1.

Property 1 guarantees that any input that satisfy the conditions in C_{min} is going to result in a failure, and the failure is going to be the failure of interest f. Property 2 states that C_{min} is 1-minimal [2], that is, every single constraint c in C_{min} is relevant (*i.e.,* necessary for reproducing the failure).

We further consider C_{min} as consisting of two types of conditions: *enabling conditions* and *failure conditions*. That is, $C_{min} = C_{enabling} \cup C_{fail}$. $C_{enabling}$ is the set of all constraints in C_{min} that are necessary to make the input valid and reach the point of failure, but are not necessary for triggering the fault of interest f. More formally:

$$\forall c \in C_{enabling}, \forall i \in I \text{ that satisfies } C_{min} - \{c\} : \\ p(i) \neq orac(i), p(i) \neq o_f$$

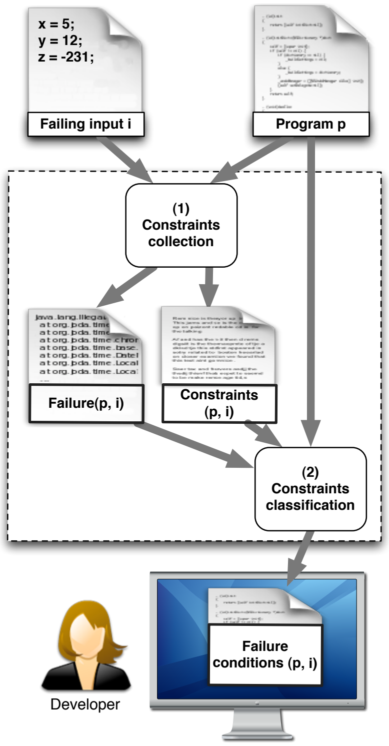

Fig. 2. Intuitive view of our approach.

C_{fail}, the set of constraints that are actually necessary to trigger the failure of interest f is then simply defined as the difference between the two sets of minimal conditions and enabling conditions: $C_{fail} = C_{min} - C_{enabling}$

The goal of our technique is therefore the following: given a program p, an oracle *orac*, and a failure producing input i_f, compute the set of enabling conditions $C_{enabling}$ and failure conditions C_{fail}. In the next section, we present our approach and show how it would work for the failure of the credit card example discussed in the Introduction.

III. Our Approach

As stated in the Introduction, the basic idea behind our approach is to *automatically* infer the conditions under which a program fails. Intuitively, such conditions would represent an explanation of the failure and could help developers understand, locate, and fix the fault(s) causing the failure. Figure 2 provides an overview of the approach.

As the figure shows, our technique takes two inputs—a program p and an input i that makes the program fail—and produces one output—a set of *failure conditions* that can be examined by the developer. To produce such conditions, the technique performs two steps: (1) constraints collection and (2) constraints classification. In the rest of this section, we first provide more details on the motivating example that we presented in the Introduction and then describe the two steps of the technique in detail using the example.

A. Motivating Example

As a motivating example, we use the `processCard` application from the Introduction, a program previously used

```
Program p:

   boolean isValidCardNumber(String ccn) {
 1.  if(ccn.length() != 16) return false;
 2.  int sum = 0;
 3.  boolean alternate = false;
 4.  int i = ccn.length() - 1;
 5.  for (; i >= 0; i--) {
 6.    int n = mapChar(ccn.charAt(i));
 7.    if (alternate) {
 8.      n *= 2;
 9.      if (n > 9) n = (n % 10) + 1;
10.    }
11.    sum += n;
12.    alternate = !alternate;
13.  }
14.  return (sum % 10) == 0;
   }

   void prettyPrintCardNumber(String ccn) {
15.    //pretty print card number
   }

   void processCard(String ccn) {
16.  if(ccn.startsWith("4"))
17.    //process Visa and exit
18.  else if(ccn.charAt(0) == '3')
19.         if(ccn.charAt(1) == '4')
20.           if(ccn.charAt(2) == '7')
21.             //process American Express and exit
22.           else
23.             throw new InaccurateCardType("AmEx:" + ccn);
24.  else if(ccn.charAt(0) == '6')
25.         if(ccn.charAt(1) == '5')
26.           if(ccn.charAt(2) == '0')
27.             //process Discover and exit
28.           else
29.             throw new InaccurateCardType("Discover:" + ccn);
30.  throw new UnknownCardType(ccn);
   }

   int mapChar(char c) {
31.  return (c >= '0' && c <= '9') ? c-'0' : c-'A'+10;
   }

   void main(String[] args) {
32.  if(isValidCardNumber(args[0])) {
33.    prettyPrintCardNumber(args[0]);
34.    processCard(args[0]);
     }
   }
```

Input used by test t: `6510 2556 8418 3585`

Fig. 3. Code excerpt and input for our motivating example.

by one of the authors in related work [3], whose source code is shown in Figure 3. The credit card number given as argument is first passed to `isValidCardNumber()`, which checks whether the number is valid. If the credit card number is valid, the program calls function `prettyPrintCardNumber()`, not shown in the figure, which performs some formatting of the card number based on the value and position of the digits. The main function then invokes `processCard()`, which checks the number's prefix to determine the type of the credit card number (*i.e.,* Visa, American Express, or Discover) and processes the card accordingly. If the prefix does not correspond to any of the supported cards, the program throws an `UnknownCardType` exception. If the prefix is similar, but not identical, to that of a supported card, the program throws an `InaccurateCardType` exception, which contains information on the card with a similar prefix. (Note that, in the example, we have made explicit the nested `ifs` that would appear when the code is compiled, which helps understanding how the constraints are generated by our approach.) Figure 3 also shows a test input that makes the program fail: `6521 2556 8414 3585`.

```
Constraints (p,t):
; constraints from mapChar
v0 ≥ '0' ∧ v0 ≤ '9' ∧
...
v15 ≥ '0' ∧ v15 ≤ '9'
; constraints from prettyPrintCardNumber
...
; constraints from isValidCardNumber
((v0 - '0') * 2) > 9 ∧ ((v2 - '0') * 2) ≤ 9 ∧
((v4 - '0') * 2) ≤ 9 ∧ ((v6 - '0') * 2) > 9 ∧
((v8 - '0') * 2) > 9 ∧ ((v10 - '0') * 2) ≤ 9 ∧
((v12 - '0') * 2) ≤ 9 ∧ ((v14 - '0') * 2) > 9 ∧
((((((v0 - '0') * 2) % 10) + 1) + ((v1 - '0') + (((v2 - '0')
* 2) + ((v3 - '0') + (((v4 - '0') * 2) + ((v5 - '0') +
(((((v6 - '0') * 2) % 10) + 1) + ((v7 - '0') + (((((v8 - '0')
* 2) % 10) + 1) + ((v9 - '0') + (((v10 - '0') * 2) + ((v11 -
'0') + (((v12 - '0') * 2) + ((v13 - '0') + (((((v14 - '0') *
2) % 10) + 1) + (v15 - '0')))))))))))))))) % 10) = 0
; constraints from processCard
v0 ≠ '4' ∧ v0 ≠ '3' ∧ v0 = '6' ∧ v1 = '5' ∧ v2 ≠ '0'
```

Fig. 4. Constraints computed for our motivating example.

B. Phase 1: Constraints collection

Phase 1 of our technique first runs the given test to detect the specific failure under investigation and collect the PC corresponding to the failure. As stated in Section II, the PC consists of a set of constraints on the input parameters. In the pseudocode shown in Figure 5, which depicts the main steps of our technique, Phase 1 is summarized by lines 1–3 of the algorithm, where `executeConSym` is the function that executes the program concretely and symbolically at the same time. The constraints that would be gathered for our motivating example are shown in Figure 4. As the figure shows, the constraints can be fairly complex even for a relatively simple and short fragment of code.

C. Phase 2: Constraints classification

After collecting the PC for the failing execution of interest, in Phase 2 our technique iterates over the set of constraints in the PC to classify them as irrelevant, enabling, or failure conditions (see Section II). As shown in the pseudocode in Figure 5, for each constraint c, the technique (a) creates a PC' where c is replaced with its negation and (b) tries to get a solution for PC' (*i.e.,* a set of input values that satisfy PC') using a constraint solver.

If PC' is unsatisfiable, or the constraint solver timeouts trying to find a solution for it, the algorithm delays the classification of c (not shown in the pseudocode for simplicity) until some other constraints are dropped. (In future work, we will also consider the possibility of having the constraint solver return the minimal unsatisfiable core and negate all related constraints at once.) Conversely, if PC' is satisfiable, our technique reruns the program using the newly computed input values, which can result in one of three outcomes:

1) The new inputs still cause the original failure (line 8), which means that c did not affect the outcome of the execution. In this case, the technique classifies c as irrelevant and drops it.

```
Parameters: program $p$, failing input $i$
Result: failure conditions $C_{fail}$, enabling conditions $C_{enabling}$
 1: $C_{fail} := true; C_{enabling} := true$;
 2: $o := p(i)$;
 3: $PC := executeConSym(p, i)$;
 4: for all $c \in PC$ do
 5:     $PC' := PC - \{c\} + \{\neg c\}$;
 6:     $i' := solve(PC')$;
 7:     $o' := p(i')$;
 8:     if $o = o'$ then
 9:         $PC := PC - \{c\}$;
10:     else if $orac(i') \neq o'$ then
11:         $C_{enabling} := C_{enabling} \wedge \{c\}$;
12:     else
13:         $C_{fail} := C_{fail} \wedge \{c\}$;
14:     end if
15:     $summarizeConstraints(C_{enabling}, C_{fail})$;
16: end for
```

Fig. 5. Simplified pseudocode that depicts our approach.

2) The new inputs cause a failure different from the original one (line 10). In this case, we assume that the constraint represents a precondition that the inputs must satisfy to be valid and reach the point of the original failure. Thus, our technique classifies c as an enabling condition.
3) The new inputs trigger neither the original nor a different failure (line 12). In this case, our technique classifies c as a failure condition, that is, a conditions that is necessary to trigger the failure of interest.

At the end of Phase 2, our technique has identified a subset of the original PC divided into two sets of constraints: enabling and failure conditions. For our motivating example, the classified constraints are shown in Figure 6. In the figure, the constraints produced by `prettyPrintCardNumber()` are missing, as they are identified as irrelevant by our approach.

Finally, to summarize the constraints, our technique uses information collected during dynamic symbolic execution to (a) identify and factor out recurring expressions within constraints and (b) identify methods that return a boolean value and can be used to summarize subsets of constraints they generate (line 15). For our example, the technique would identify the constraints created while executing `isValidCardNumber()` as constraints generated by a boolean method and would suitably replace them with the name of the method. The result of such substitution would be the short and crisp description of the failure that we showed in the Introduction and repeat here:

$$isValidCardNumber(v) \wedge v_0 = \texttt{'6'} \wedge v_1 = \texttt{'5'} \wedge v_2 \neq \texttt{'0'}$$

As this description would be generated in a fully automated way, it could be provided to the developers at no additional (human) cost to help them understand and eliminate the failure.

IV. Simplifying Constraints

To illustrate constraint simplification in more depth, consider the code in Figure 7. Assume we also have a test case that calls method `doSomething()` with a value 4 for parameter x. The initial step of our algorithm of Figure 5 would execute the test and observe an `IllegalStateException` being thrown. The corresponding PC would consist of the following set of constraints: $(x \geq 0) \wedge (x \leq 5) \wedge (x < 7)$.

```
Enabling conditions (p, t):
; constraints from mapChar
v0 ≥ '0' ∧ v0 ≤ '9' ∧
...
v15 ≥ '0' ∧ v15 ≤ '9'

Failure conditions (p, t):
; constraints from isValidCardNumber
((v0 - '0') * 2) > 9 ∧ ((v2 - '0') * 2) ≤ 9 ∧
((v4 - '0') * 2) ≤ 9 ∧ ((v6 - '0') * 2) > 9 ∧
((v8 - '0') * 2) > 9 ∧ ((v10 - '0') * 2) ≤ 9 ∧
((v12 - '0') * 2) ≤ 9 ∧ ((v14 - '0') * 2) > 9 ∧
((((((v0 - '0') * 2) % 10) + 1) + ((v1 - '0') + (((v2 - '0')
* 2) + ((v3 - '0') + (((v4 - '0') * 2) + ((v5 - '0') +
(((((v6 - '0') * 2) % 10) + 1) + ((v7 - '0') + (((((v8 - '0')
* 2) % 10) + 1) + ((v9 - '0') + (((v10 - '0') * 2) + ((v11 -
'0') + (((v12 - '0') * 2) + ((v13 - '0') + (((((v14 - '0') *
2) % 10) + 1) + (v15 - '0')))))))))))))))) % 10) = 0
; constraints from processCard
v0 ≠ '4' ∧ v0 ≠ '3' ∧ v0 = '6' ∧ v1 = '5' ∧ v2 ≠ '0'
```

Fig. 6. Classified constraints for our motivating example.

The technique would then iterate over the constraints to classify them. Assume that the algorithm arbitrarily choses $(x \geq 0)$ to be classified first. Negating it yields the following PC': $(x < 0) \wedge (x < 5) \wedge (x < 7)$. A constraint solver fed with these constraints would return a value of x that satisfies the constraints, such as $x = -1$. The technique would then execute the test again with parameter x being -1, which would result in an `IllegalArgumentException` being thrown. Because this failure is different from the initial one, our algorithm would classify the constraint that was negated as an enabling condition and keep it in the PC. In fact, this constraint encodes the condition for an input to be valid and, although it does not trigger the failure, it is necessary to reproduce it.

Assume now that the next constraint the algorithm choses to negate is $(x \leq 5)$, which results in the PC' being $(x \geq 0) \wedge (x > 5) \wedge (x < 7)$. Given these constraints, the constraint solver would return $x = 6$. Again, the technique would rerun the program, this time setting parameter x to 6. The outcome of this execution would be an `IllegalStateException`, which is the original failure we are analyzing. The technique would therefore conclude that the constraint is irrelevant for reproducing the failure and would drop it, leaving only $(x \geq 0) \wedge (x < 7)$ in the set of

```
public void doSomething(int x){
  if (x < 0) {
    throw new IllegalArgumentException(...);
  }
  if (x <= 5) {
    System.out.println("x is " + x);
  }
  if (x < 7) {
    throw new IllegalStateException(...);
  }
}
```

Fig. 7. Simple example to further illustrate our approach.

constraints. Again, it is clear from the code that this is the right course of action, as the condition guards a statement that writes to the console; for the failure to occur, it is irrelevant, whether that statement is executed or not.

The next and last constraint the algorithm choses to negate would be $(x < 7)$, which results in PC' being $(x \geq 0) \wedge (x \geq 7)$. In this case, any value for x greater or equal to 7 would satisfy the constraints, so let us just assume that the constraint solver produces the solution $x = 7$. When the technique reruns the program with x being 7, the result is a normal execution of the program that does not produce any failure. The algorithm would therefore classify the constraint as a failure condition, a constraint that is necessary for the failure of interest to be triggered.

At the end, the technique would present the developer with two constraints: constraint $(x \geq 0)$, marked as an enabling condition, and constraint $(x < 7)$, marked as a failure condition. The developer could then use this information to understand and correct the fault in the code.

V. A Real-world Fault

We have developed a prototype tool that implements our technique and are currently using it on a set of real-world subjects. The prototype is built on top of Java PathFinder (JPF) [4], a symbolic execution engine for Java programs, and leverages JPF's extensions for dynamic symbolic execution and the CVC3 constraint solver [5]. In this section, we share some of our initial experiences with the tool.

As a real-world example, consider Figure 8, which depicts a failing test case for the JODA TIME date and time library. This test case illustrates an issue with the `toInterval()` method. When invoked for the date October 18th, 2009, `toInterval()` abruptly terminates and raises an `IllegalArgumentException`.

An investigation of the failure revealed that the problem occurs on days on which the daylight savings time starts or ends (*i.e.,* cutover days) if the cutover time is midnight. This is also why the failure occurs in the Brazilian timezone, as in Brazil, such cutover time happens to be midnight. (The underlying reason for this failure is that method `toInterval()` should convert the `LocalDate` object to an interval representing the whole day. If for a daylight cut day the cutover time is midnight, the time 00:00:00 does not exist, and an internal consistency check raises the `IllegalArgumentException`

```
public class JodaTest extends TestCase {

  public void testBrazil() {
    DateTimeZone dtz =
      DateTimeZone.forID("America/Sao_Paulo");
    LocalDate date =
      new LocalDate(2009, 10, 18);
    Interval interval = date.toInterval(dtz);
  }
}
```

Fig. 8. JODA TIME bug 2487417: `toInterval()` fails when processing 18th of October, 2009 Brazil time.

we observe.) For this failure, our approach would ideally return the following failure condition:

$$\mathit{isDaylightCutDay}() \wedge (\mathit{cutoverTime} = \mathit{00{:}00{:}00})$$

This is a good example for illustrating the complexity of the problem we are tackling and the relevance of our approach. Although the test case in Figure 8 is fairly small and compact, it executes 367 methods and a total of 5,711 individual statements. (Note that these figures consider only code that belongs to JODA TIME and ignore the execution of code in the JAVA system libraries.) When we apply our approach to the test case, the approach generates 89 initial constraints, some of which contain around 2,000 subexpressions. In general no human could read and understand this amount of information.

Although our prototype cannot currently fully analyze this failure, due to limitations in its implementation, we were able to get some initial evidence of the potential usefulness of the approach. For example, the analysis of the constraints for identifying (a) recurring expressions and (b) boolean methods that can be used to abstract away subsets of conditions was able to dramatically reduce the complexity of the constraints (*e.g.,* some expression appeared hundreds of times in the PC). The main reason why we cannot yet completely analyze this program is the presence of mathematical operators that are not supported by CVC3, the constraint solver we use. We are currently considering the use of a different solver and also the replacement of constraints that the constraint solver cannot handle with concrete values produced during the previous execution (see Section VI).

VI. Challenges and Future Work

In this section, we discuss the main challenges we must address to improve our technique and tool.

1) Hard-to-solve Constraints: Some generated constraints go beyond the solving capabilities of existing constraint solvers (*e.g.,* constraints generated by caching and hashing functions that make use of bit-wise operators). As discussed above, we are considering two (possibly complementary) directions: the use of solvers that operate on formulas defined over the theory of bit-vectors and arrays, such as STP [6]; and the replacement of constraints that go beyond the theories supported by the solver with their concrete values from a previous execution. Our future investigation will allow us to assess how relevant and common these issues are and how to best address them.

2) Disjunctions: One problem we encountered for the initial formulation of our approach is related to the presence of predicates containing `OR` operators in the code. Since Java short-circuits `OR` operators (*i.e.,* if the first condition is true, the other conditions are not evaluated), our technique may only observe a subset of a disjunctive predicate at runtime. This is an issue because, as discussed in Section III, our technique negates constraints and drops them if the failure still occurs for inputs that satisfy the negated constraints. Consider, for instance, the code snippet shown in Figure 9, and assume that the original value of x is smaller than that of y.

```
public void do(int x, int y) {
  if (x < y || x > 10) { fail(); }
}
```
Fig. 9. Example of predicate that contains an `OR` condition.

When executing the code, the second part of the predicate would not be evaluated, and our technique would only add to the PC constraint $(x < y)$. Then, when it negates that constraint and solves the modified PC, the technique might by chance obtain from the constraint solver a value for x that is greater than 10 (*e.g.,* due to additional conditions on y). In such a case, although the constraint is violated, the predicate it belongs to (*i.e.,* $(x < y) \vee (x > 10)$) would still evaluate to true. The technique would therefore erroneously classify the constraint as irrelevant and drop it. We are currently evaluating different alternative solutions to this issue.

3) Limitations of Symbolic Execution: Because our approach relies on symbolic execution, and more specifically on dynamic symbolic execution, it also suffers from some of the limitations of this type of techniques. In particular, there may be cases where the inputs that cause the failure involve complex interactions with the environment, such as access to the network, the file system, and external databases. These types of interactions are notoriously problematic for symbolic execution techniques and may limit the general applicability of our approach.

VII. Related Work

Because debugging is a vast and active research area, due to space reason we only focus on the approaches that are mostly related to ours.

Delta Debugging [2] automatically simplifies failure-inducing circumstances by means of systematic experiments. Applied to a program input, it retains only those parts of the input necessary for producing the failure. In a way, our approach can be seen as a generalization of delta debugging, as it generalizes (or "simplifies") the constraints that govern the failure-inducing input; in fact, we are currently investigating the use of delta debugging as an additional way to simplify constraints.

Groce and colleagues [7] research the hypothesis that the comparison of a failing execution with minimally different successful executions provides information on the location and causes of an error. Based on the results of this comparison, their technique creates a dynamic slice and presents the corresponding code to the developers as the point where the defect is likely to be located. The main limitation of their approach is that it needs manual guidance by someone with knowledge of the program and its functionality. Our approach, conversely, is automated and aims more at providing an explanation of the failure than the location for the corresponding fault(s).

Clause and colleagues [3] use dynamic symbolic execution on field failures to generate failure-inducing inputs that cause the same failures while preserving user privacy. Their approach is similar to ours, as they also gather PCs and solve them to produce alternative inputs. However, while for them this alternative input is the overall goal, for us it is only an intermediate result, used to further analyze the execution and classify the resulting constraints in order to reduce them.

The approach of Qi and colleagues [8] focuses on programs that undergo a modification, such that they have two versions of the same program that differ at some points in the code. Given a failure that occurs in one of the two versions, but not in the other, they gather the PCs for the failure-inducing input in both versions of the program. Using these PCs, they then generate new inputs that follow the same execution path in the correct version and a different path in the failing version. Finally, they compare the two execution paths and present differences as possible locations of the defect. Although related, our technique operates on a single version of the program and focuses on characterizing failure circumstances rather than localizing defects.

VIII. Conclusion and Consequences

When a program fails, it is crucial to identify the circumstances under which the failure occurs. In this paper, we have illustrated this problem and presented an initial approach that aims to identify these circumstances automatically. Our approach operates by collecting the path condition of the failing execution of interest and simplifying and abstracting them using a combination of constraint manipulation and test input generation. Although we are still at the early stages of this research, we believe our preliminary results are encouraging; in the future, whenever a test fails, the programmer may not only know that it failed, but also under which circumstances it failed. These failure circumstances would give precious hints on why the failure occurred and how to avoid or eliminate it, and would represent another promising step towards reducing the burden of debugging.

Acknowledgments

This work was supported in part by NSF awards CCF-0725202 and CCR-0209322 to Georgia Tech.

References

[1] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 344–353. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.45

[2] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 183–200, February 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=506201.506206

[3] J. Clause and A. Orso, "Camouflage: Automated anonymization of field data," in *Proceedings of the Intenational Conference on Software Engineering (ICSE 2011)*, May 2011, to appear.

[4] K. Havelund and T. Pressburger, "Java pathfinder, a translator from java to promela," in *Theoretical and Practical of SPIN Model-Checking*, 1999.

[5] C. Barrett and C. Tinelli, "Cvc: A cooperating validity checker," in *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification*, 2002, pp. 500– 504.

[6] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification (CAV '07)*. Berlin, Germany: Springer-Verlag, July 2007.

[7] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 229–247, 2006.

[8] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, "Darwin: an approach for debugging evolving programs," in *ESEC/SIGSOFT FSE*, H. van Vliet and V. Issarny, Eds. ACM, 2009, pp. 33–42.