Symbolic code execution: a powerful approach for test generation

Arnaud Dury, Ph.D.
Researcher
Advanced Software Modeling and Development

December 16, 2013

arnaud.dury@crim.ca
WWW.CRIM.CA/MODL

Financial partner:
# TABLE OF CONTENTS

1. ABSTRACT .......................................................................................................................... 3

2. INTRODUCTION ................................................................................................................. 3

3. WHAT IS A SYMBOLIC CODE EXECUTION TOOL? ............................................................. 3

4. HOW DOES SUCH A TOOL WORK? .................................................................................... 4

5. EXISTING OPEN SOURCE TOOLS AND INDUSTRIAL SUCCESS STORIES ....................... 7

6. OTHER APPLICATIONS ....................................................................................................... 8

7. AREAS OF RESEARCH .................................................................................................... 8
   7.1 Combinatorial explosion ......................................................................................... 8
   7.2 Memory modeling ................................................................................................. 8
   7.3 External libraries .................................................................................................... 9
   7.4 Concurrency .......................................................................................................... 9

8. CONCLUSION .................................................................................................................... 9

9. REFERENCES ................................................................................................................... 10
1. **ABSTRACT**

Symbolic code execution is an automated technique capable of generating tests with high coverage in complex software applications. The rigorous analysis it performs can often detect subtle errors missed by other approaches. The technique started as a concept in the seventies, and progressed to usable prototypes during the last fifteen years. Applicability for test generation on non-trivial applications has been successfully demonstrated, outperforming the long-term manual testing efforts of the whole teams building those applications. On a test suite of 89 stand-alone programs, forming the core of the user-level environment of UNIX systems, the technique found three serious bugs which had been missed over 15 consecutive years of testing. Important applications of the technique in other areas of software development such as software verification, impact change analysis, input sequence synthesis, and software security are also demonstrated. The paper explains the main principles of the technique and its application to test generation, presents some of the successful applications, the limitations of the existing tools and the research directions currently being explored.

2. **INTRODUCTION**

Symbolic code execution is an automated technique capable of generating tests with high coverage in complex software applications. The rigorous analysis it performs can often detect subtle errors missed by other approaches. Important applications of the technique in other areas of software development such as software verification, impact change analysis, input sequence synthesis, and software security are also demonstrated. It was first proposed in the seventies [5] but then abandoned: the computers at the time were not powerful enough to support the technique. Interest has been rekindled in the last fifteen years and several prototype tools have been developed and have demonstrated their applicability in real-world settings [4]. This white paper presents the main ideas behind the technique and its inherent deep analysis, the recent industrial successes, a summary of other fields of software development to which it can contribute, and finally an overview of the remaining challenges and current research efforts.

3. **WHAT IS A SYMBOLIC CODE EXECUTION TOOL?**

A symbolic code execution tool is a *specialized interpreter* that can execute a program in a given language over both concrete and symbolic inputs. The language can be either a high-level one like C, C++, Java or JavaScript or low-level ones like bytecodes and binary codes. To understand the difference between concrete and symbolic inputs, consider a computer program that accepts three integers as input, that performs a set of computations, and that outputs a result depending on those. The usual, or *concrete*, execution of the program will be performed on a specific valuation of these three integers, for instance 42, 128, and 897. Since the integers provided as inputs are defined prior to the program execution they are called *concrete inputs*, and the program execution is called a *concrete execution*.

On the contrary, symbolic code execution executes a program over *symbolic inputs*. A symbolic input is a symbolic representation (such as “\(x\)”) of a concrete input of a program (such as 42). A *symbolic execution* of the previous program would consist of an
execution of the program over the three symbolic inputs $x$, $y$ and $z$. For simple programs, such execution can compute the symbolic outputs as a function of the symbolic inputs, but this is not always possible for a program of arbitrary size or complexity. Symbolic code execution is instead used for other interesting purposes, such as automated test generation. This presentation focuses on test generation; other usages of symbolic code execution are presented in the last section.

4. HOW DOES SUCH A TOOL WORK?

A symbolic code execution tool can accept both concrete and symbolic inputs, and processes the former as a concrete code interpreter would. The main strength of such tools is of course their ability to handle the latter. How to execute a program with symbolic inputs?

```
main()
{
   int a,b;
   scanf("%d",&a);// a<-@si
   if (a>10){
      b=2*a;
   }
   if (a<20){
      b=4*a;
   }
}
```

*Figure 1*

Symbolic program execution redefines the way an interpreter works by defining symbolic mechanisms for variable assignment, conditional statements, loops, expression evaluation, etc. The very simple example in Figure 1 is used to illustrate two important aspects of symbolic code execution: handling of branching, and handling of symbolic variable assignments.

Figure 2 depicts the control flow of this example. We now explain how symbolic execution can express the output of this example as a function of its input, and how such expression is then used to automatically generate a minimal set of tests that guarantees 100% path coverage.

The first step for the developer is to tag the input variable(s) to be considered as symbolic (such as the keyboard input in this example, tagged as a symbolic input named $si$). Symbolic program execution then starts. Line 1 is executed as it would be in a concrete execution: variables $a$ and $b$ are created and assigned with default values. At line 2, the variable $a$ is assigned with the symbolic input $si$. Execution of line 3 proceeds as follows: since $a$ is symbolic, there is no way to know whether the condition $(a>10)$ is true or false; the value of $a$ is unknown at that point, and could
be any integer value. But something can be known for sure: if program execution continues to line 4, then it is necessarily true than \( a > 10 \), and therefore that \( s_i > 10 \). Conversely, if program execution continues immediately on line 6 instead of 4, then necessarily \( a \leq 10 \) and therefore \( s_i \leq 10 \). Symbolic program execution proceeds as follows: when encountering the conditional in line 3, it explores both possible executions (the executions following the \texttt{then} and the \texttt{else} branches of the conditional) in parallel and remembers for each one the condition on \( s_i \) that must have been true on that execution.

This condition for a given execution is called its \textit{path condition}. A path condition represents what can be known about the valuation of the variables of the program at a given point, and such knowledge is expressed as a logical formula on the input values of the program (there is only one, \( s_i \), in our example). Each execution may later generate new branches if it encounters another conditional. The symbolic execution of the example on Figure 1 produces three executions in total (and not four!): the execution taking the \texttt{true} edge of the control flow graph at the first conditional, then the \texttt{true} edge at the second; the \texttt{true} edge then the \texttt{false} edge; and finally the \texttt{false} and then the \texttt{true} edge.

![Figure 2](image)

The exploration of the execution taking both \texttt{false} edges would require the following formula to be \texttt{true}: \((s_i \leq 10) \) and \((s_i \geq 20)\), which is immediately determined as being impossible. The symbolic executions that are explored are only those for which there exists at least a concrete execution satisfying the corresponding path condition. A single symbolic execution usually represents many concrete ones. Symbolic variable assignments (lines 4 and 7) are handled this way: the variable \( b \) is assigned with the symbolic expression on the right hand side of the assignment, expressed in terms of the symbolic inputs. For instance, on line 4, the variable \( b \) is assigned with the symbolic expression \( 2 \times s_i \) since at that point \( a = s_i \). When an execution terminates, the values
of a and b are expressed in terms of a function of si, and the tool can generate concrete tests (that is, concrete inputs and expected concrete output) covering the execution.

Test generation is performed by solving the path condition using a relevant solver (off-the-shelf specialized solvers called SMT solvers for “SAT Modulo Theory solvers” are generally used at this point, such as Yices[3] or Z3[2]). The solution provides concrete values for si for the execution, and the concrete output is then computed. As an example, the execution that executed the then branch of the first conditional and the then branch of the second conditional has the following path condition: (si>10) and (si<20). Solving the path condition with an SMT solver returns one possible concrete value for si satisfying this path condition. Since the input value is defined as an integer, the solver returns a concrete integer value between 11 and 19. Let’s assume that it returns si=11.

The symbolic execution corresponding to this path condition terminates with the symbolic value for b: b=4*si. A proper concrete test for this execution is therefore defined by the input si=11 and the expected output b=4*si is 44. Notice that while this program accepts 232 concrete executions (one for each concrete value of a, assuming a 32 bits representation for integers), it still has only three symbolic executions, each of them summarizing a high number of concrete ones. While exhaustive exploration of the space of concrete executions (testing the program with all possible concrete input values) would be too costly, exhaustive exploration of the space of symbolic execution for this program is not.

A more interesting example is shown in Figure 3. In this example, computations are performed on a symbolic variable inside a loop. The code after the loop contains an instruction that can raise an exception at runtime: the division of b by a on line 16. A tester wants to know the answer to the following question: is there a particular input for a on line 2 that could lead this program to crash on a division by zero on line 16? In order to test for such possibility, the program is given to a symbolic code analyzer.
The analyzer used here is a prototype we are currently developing at CRIM. It detects 27 potential symbolic executions and determines that only seven of them are feasible (the path condition cannot be satisfied for the remaining 20, they therefore accept no corresponding concrete executions and none of them are explored further). It also proves that only one of the seven leads to a division by zero: the one starting with the non-obvious input -14. Not only do we now know that it exists only one input value leading to a division by zero, we also know the input value generating such exception. This small example illustrates the power of the technique: it is entirely automated, can detect runtime errors with perfect precision, and does not rely on the exhaustive generation of all possible inputs.

5. **EXISTING OPEN SOURCE TOOLS AND INDUSTRIAL SUCCESS STORIES**

Tools for symbolic code execution have been produced by research groups around the world, and some of them are open source (such as KLEE[4], CREST[6], JPF-SE[7]). At this stage they can be considered as research prototypes, usable by power users willing to deal with web forums and limited documentation to find help on installation, configuration and usage. One of the most advanced prototypes for the C language is KLEE (available at http://klee.llvm.org) which has been successfully applied to C projects ranging from network servers and tools, to UNIX file systems, utilities and drivers, exploit generation and wireless sensor networks among others. KLEE was tested on the GNU CoreUtils utility suite totalling 89 stand-alone programs, forming the core of the user-level environment of UNIX systems. This suite has been extensively used by numerous users, and finding previously unknown bugs is a difficult task. KLEE generated a test suite achieving over 90% line coverage, well over the manual test
suite coverage obtained by the developers over the whole development time of the tools suite. Moreover, three serious bugs were discovered which had been missed over 15 consecutive years of testing. On three other similarly comprehensive tool suites, KLEE found a total of 56 serious bugs. Other symbolic code execution tools have been successfully applied in the domains of execution synthesis of multi-threaded programs [8], the reverse engineering of binary device drivers [9] and the testing of NASA space networks [10].

6. OTHER APPLICATIONS

Symbolic code execution is explored for other purposes than test generation. It can be used with a user-defined property on program states to determine whether the property holds or not; to analyze a modified version of a given program and compare its behaviour with respect to the original version (change impact analysis for instance, or comparison of two implementations in different languages); to generate an input sequence leading a program from a given global concrete state to another (for test repair for instance); to automatically build input filters preventing crashes in a given application (such as divide by zero, or memory access outside of array bounds) or conversely to build an input sequence leading to an exploit in the code; etc.

7. AREAS OF RESEARCH

Symbolic code execution is an active research and development field. The main research areas are related to combinatorial explosion, memory modelling, usage of external libraries and concurrency.

7.1 Combinatorial explosion

Generating new execution at each decision point can lead to a combinatorial explosion (and even a possibly infinite number) of explored executions, for large and complex programs. The state of the art of the research focuses on the analysis of such programs function by function and later combination of the results; combination of concrete and symbolic execution; use of heuristic to efficiently explore the execution space; SMT formula simplification and results caching; efficient state representation; and state merging [1].

7.2 Memory modeling

The modeling of the memory used by a program is also a research topic, particularly for languages allowing the use of pointers like the C language. A pointer in C could reference, in the general case, any memory location accessible by the program. If the pointer is treated as symbolic, it would in turn forces symbolic execution to consider as many potential executions as there are accessible memory locations. Since this is intractable in practice, various techniques are elaborated to allow symbolic execution to perform reasonably on code using pointer arithmetic. The main idea is to restrict the usage of symbolic pointers to specific cases which can be still analyzed symbolically, and to build memory representation which can be efficiently reasoned upon.
7.3 External libraries

The use of external libraries for which the source code is not accessible can also be an issue, since symbolic execution assumes the source code is available. Proposed approaches cover the construction of models abstracting the behaviour of library functions, the combination of concrete and symbolic execution, and the reverse engineering of code from binary executable.

7.4 Concurrency

Finally, symbolic analysis of applications composed of several concurrent programs is currently studied by a merging of classical model checking techniques (to analyze the interleaving of actions produced by concurrent programs) with symbolic execution techniques which focus on the behaviour of each one.

8. CONCLUSION

Symbolic code execution started as a concept in the seventies, and progressed to usable prototypes during the last fifteen years. Applicability for test generation on non-trivial applications has now been successfully demonstrated, outperforming in an automated way the long-term manual testing efforts of the whole teams building those applications. While few Open Source tools are currently available (and most of them are research prototypes rather than user friendly tools), research and development is progressing constantly and such tools may be included in future Open Source IDE. CRIM is becoming very active in further developing this technology.
9. REFERENCES


