Specification Extraction by Symbolic Execution

Josef Pichler
Software Competence Center Hagenberg
4232 Hagenberg, Austria
josef.pichler@scch.at

***Abstract*—Technical software systems contain extensive and complex computations that are frequently implemented in an optimized and unstructured way. Computations are, therefore, hard to comprehend from source code. If no other documentation exists, it is a tedious endeavor to understand which input data impact on a particular computation and how a program does achieves a particular result. We apply symbolic execution to automatically extract computations from source code. Symbolic execution makes it possible to identify input and output data, the actual computation as well as constraints of a particular computation, independently of encountered optimizations and unstructured program elements. The proposed technique may be used to improve maintenance and reengineering activities concerning legacy code in scientific and engineering domains.**

***Index Terms*—Legacy code, symbolic execution, reverse engineering, knowledge extraction.**

I. Introduction

Many technical software systems contain extensive and complex computations that are hard to comprehend. Fortran libraries such as BLAS[1], CERNLIB[2], or SPICE[3] are typical examples of such software systems. But also companies in various engineering domains such as electrical engineering or process engineering run and maintain programs that can be characterized by the following properties [1]:

- complex and extensive computations
- optimizations (running time, memory requirements)
- frequent changes (lessons learned from the field)

Optimizations with respect to running time or memory requirements are often the most important quality aspects compromising modularity and structure of the software. As a consequence, such software systems are hard to comprehend, in particular in case of legacy code, when one is faced with the problem of using the software without the support of the programmers who designed or implemented it.

If no other documentation exists and the source code becomes the only source of domain knowledge, it is hard to understand how the software calculates output depending on which input. In particular, answering following questions is a hard endeavor:

1. Which parameters have impact on a particular computation?
2. How does the program achieve a particular result?
3. What are the constraints of a particular computation?

[1] http://www.netlib.org/blas/
[2] http://cernlib.web.cern.ch/cernlib/
[3] http://naif.jpl.nasa.gov/naif/toolkit.html

Reverse engineering tools [2, 3] are a good starting point to answer such questions. Reverse engineering tools support software developers to reason about a software system by producing high-level representations in form of call graphs, program dependency graphs, dataflow graphs, class diagrams, sequence diagrams, etc. Such representations are useful to answer question 1 but have limited merit for finding answers to the other questions. Model extraction tools [1, 4, 5, 6] go one step further and produce higher-level representations such as business rules. However, such tools are mostly based on pattern-matching or transformation rules and, therefore, can be applied to structured code only.

Symbolic execution introduced by King [7] and Clark [8] explores program behaviors along as many execution paths as possible. Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be *symbolic formulas* over the input symbols [7]. Compared to traditional static analysis such as dataflow analysis, symbolic execution tracks actual symbolic values and relations of variables at program points along program paths. Symbolic execution outperforms static analysis techniques through being applicable to unstructured code.

In this paper, we propose an approach to extract specifications from source code by means of symbolic execution. Our idea is to exploit the resulting symbolic formulas over the input symbols to deduce specifications from given program code. Symbolic execution makes it possible to identify input and output data, the actual computation as well as constraints of a particular computation, independently of the program structure.

The contributions of this paper are:

- to exploit symbolic execution for extraction of specifications from source code, and
- an elementary tool implementation to demonstrate the feasibility of the proposed approach.

The rest of this paper begins with a brief discussion of the problem we want to address (Section II) before detailing the overall idea and the technical aspects of our approach and tool (Section III) and presenting its power and limitations by two examples in Section IV. Related work (Section V) and our conclusions (Section VI) follow.

II. The Problem Statement

We illustrate the problem by a simple but meaningful example that contains representative characteristics of source code, for which our approach is intended for. Consider the

978-1-4799-2931-3/13/$31.00 © 2013 IEEE

program code of routine RX given in Listing 1, which computes the equivalent resistance R_EQ of a series of resistors R connected in parallel. The code is given in a Pascal-like pseudo code [9].

```
RX(↓R: array[1:N] of real ↓N: int ↑R_EQ: real)
  var I: int
begin
  R_EQ := R[1]
  if N = 2 then
    R_EQ := R_EQ*R[2]/(R_EQ + R[2])
  else if N > 2 then
    R_EQ := 0
    for I:=1 to N do
      R_EQ := R_EQ + 1/R[I]
    end
    R_EQ := 1/R_EQ
  end
  end
end RX
```

Listing 1. Routine to compute the equivalent resistance R_EQ of N resistors R connected in parallel.

Equation 1 is the expected specification of routine *RX* including three different cases depending on the number *N* of resistors. Besides the common formula for case $N > 2$, the specification additionally contains a simplified case for $N = 1$ and $N = 2$. This is for performance reasons only, because the common formula is a valid solution for $N = 1$ and $N = 2$ as well. This pattern frequently appears in optimized program code of scientific and engineering software system.

$$R_{EQ} = \begin{cases} R_1 & N = 1 \\ \frac{R_1 \cdot R_2}{R_1 + R_2} & N = 2 \\ \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \cdots + \frac{1}{R_N}} & N > 2 \end{cases} \quad (1)$$

Even the program code uses structured elements only (i.e. no *goto* statement) and we assume that we already know the objective of the routine *RX* and the input and output data, it remains a hard endeavor to precisely answer questions 2 and 3 concerning how the function achieves a particular result and under which constraints. Based on the given example, we identify following challenges for understanding the program code and extracting the specification in the expected form.

1. The constraint $N = 1$ for the computation $R_{EQ} = R_1$ does not appear in the program code but can be deduced from the conditions $N = 2$ and $N > 2$ only. In general and in case of unstructured code, this is a non-trivial task.
2. The computation for the constraint $N = 2$ is implemented in two statements whereas the second one is additionally surrounded with a conditional statement:

   ```
   R_EQ := R[1]
   R_EQ := R_EQ*R[2]/(R_EQ + R[2])
   ```

   The reuse of R_EQ in the second line instead of accessing R[1] is a frequent pattern to reduce the number of array accesses.
3. The *for* loop statement together with the initialization R_EQ := 0 actually computes the sum over $1/R_i$. The same pattern can be identified for computation of other aggregate functions such as product, min, and max.
4. The variable R_EQ contains different domain concepts in different lines. Within the *for*-loop statement, R_EQ is the sum of the conductance, whereas at all other places it is the equivalent resistance.

In the next section, we describe how we can apply symbolic execution to extract expected specifications from program code with the described characteristics.

III. Extraction By Symbolic Execution

A. The Idea

The central idea of our technique is to perform symbolic execution of the program fragment of interest and to collect the *symbolic formulas* together with the so called *path conditions*. A symbolic formula of an output parameter facilitates answers to question 2 "*How does the program achieve a particular result?*". The path condition defined by King [7] as "*the accumulator of properties which the inputs must satisfy in order for an execution to follow the particular associated path*", answers question 3 "*What are the constraints of a particular computation?*". Both the symbolic formulas as well as the path condition are always expressed in terms of the program inputs and, therefore, constitute the expected specification of the program to be analyzed.

Symbolic execution typically has low scalability due to the fact that all paths of a program need to be analyzed (so called path explosion problem). In particular, condition statements, loop statements, and calls to libraries, for which no source code is available, are challenging for symbolic execution [10, 11, 12]. To overcome these challenges, we use *dynamic symbolic execution* [13] and perform a *concrete execution* of the program simultaneously with its symbolic execution. The key idea here is to start executing a program with concrete values and using these values to steer the execution of conditional statements and loop statements, when symbolically execution comes to its limit. As consequence, a single execution path specified by concrete input values is explored and symbolic formulas and path conditions are generated for every execution path. Several program executions are performed in sequence to cover all program paths of interest. The concrete input is mainly used to control the execution path in situations that are challenging for symbolic execution.

B. Overall Picture

Fig. 1 shows the overall picture of the proposed technique. The arrows show the flow of information between various components. Language parsers create ① the abstract syntax tree (AST) of the program to be analyzed. Currently, we have implemented a parser for the pseudo code used for all examples in this paper. The *Execution Engine* executes ② the AST driven by concrete input values ③ for a selected function or subroutine and collects results from symbolic execution. For a single execution specified by a set of concrete input values, this process generates ④ concrete and symbolic result. With respect to the routine *RX* given in Listing 1 and the execution path #2 illustrated in Fig. 1, the result is:

- a concrete result value (40)
- a symbolic formula ($R_{EQ} = (R_1 \cdot R_2)/(R_1 + R_2)$)
- a path condition ($N = 2$)

The symbolic expressions concerning formula and path condition represent the unmodified computation from the program code. In general, this form is more related to program code than to the expected specification. To transform results from symbolic execution towards the expected formulas of the specification, a canonical simplification ⑤ is applied to both symbolic results (simplified terms are bold and underlined in Fig. 1). Last, results of different executions of the same function are collected.

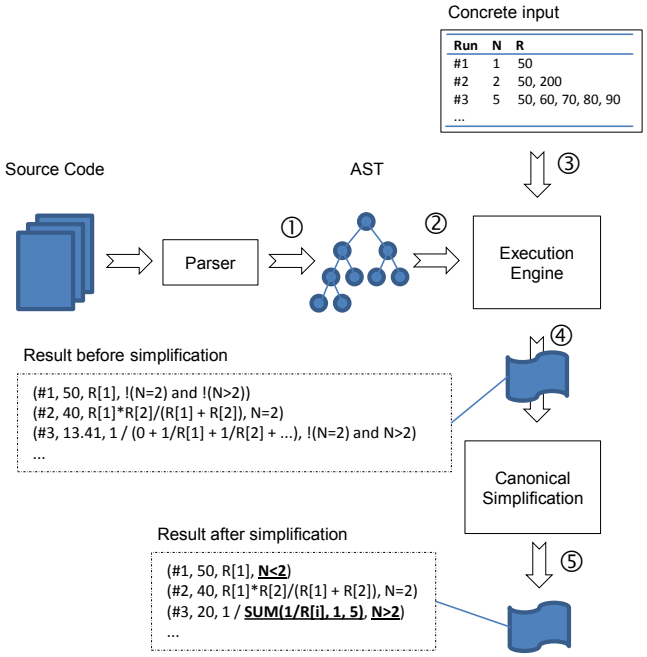

Fig. 1. Symbolic execution for formula extraction.

C. Input Data

As outlined earlier, the symbolic execution is driven by concrete input, which must be provided by the user performing the analysis. If the user has some clue about the program code under investigation, analysis is usually started with some valid input data for a common case. Otherwise, the user may select input values randomly according to the data types of input parameters. After the first execution, the path condition of the extracted formula may assist the user to find further input values for execution paths of interest and to decide whether the extracted formulas cover the specified value range of input data and, hence, whether the specification is complete. For instance, the three execution paths illustrated in Fig. 1 cover execution paths for the constraints $N < 2$, $N = 2$, and $N > 2$. If the user is interested in computations with $N \geq 1$ only, then the extracted formulas together yield the entire specification of routine *RX*.

D. Computation of Symbolic Formulas

The computation of symbolic formula takes place during execution of the program fragment driven by concrete values.

The *Execution Engine* operates on the abstract syntax tree (AST) and constructs symbolic values for program variables and also performs operators and intrinsic functions on variables and literals symbolically with respect to the semantics of the operators, functions, and data types of operands. In that way, not only concrete values are calculated and stored during calculation but also a symbolic formula together with the path condition. Currently, the tool handles classical operators and intrinsic functions of procedural languages concerning number types (both integer and floating point numbers), boolean values, as well as arrays but no pointer types or pointer arithmetic.

E. Canonical Simplification

The symbolic formulas and path conditions generated by the *Execution Engine* are further simplified in a canonical way by means of reduction rules. To simplify arithmetic and boolean expressions, we apply reduction rules such as "0 + x ⇨ x" and "x – x ⇨ 0" on numbers and "true and x ⇨ x" on boolean values. To simplify path conditions we further apply rules for relational operators on integer numbers so that we can reduce the path condition for run #1 given in Fig. 1 "not (N = 2) and not (N > 2)" to N < 2. Concerning arrays, rules are applied to reduce expressions of the form a[1] + a[2] + ... + a[10] to an aggregate function SUM(a, 1, 10), as illustrated in Fig. 1 for run #3.

IV. Examples

In this section, we apply our proposed technique on two examples to illustrate its power but also the current limitation of our preliminary tool. The purpose of the examples is to convey how symbolic execution can be used to understand how the software calculates output depending on which input and to deduce a complete specification.

A. Example: Parallel Resistances

In this example, we perform symbolic execution on the routine *RX* given in Listing 1, which is also illustrated in Fig. 1. If we execute the routine with R = (50) and N = 1, the result will be (formula and path condition are separated by |)

```
R_EQ = R[1] | N < 2
```

Note that the value 50 does not appear in the result, thus, the computation depends on the parameter N only. Whereas the derivation of the symbolic formula is trivial, the path condition was simplified from the expression `!(N=2) and !(N>2)` over integer numbers.

In the second run, we execute the routine with R = (50, 200) and N = 2 and get the result

```
R_EQ = R[1]*R[2]/(R[1] + R[2]) | N = 2
```

Both the symbolic formula and path condition correspond to the program code, except the substitutions of `R_EQ` with `R[1]`. This result corresponds to the expected specification for N = 2 without further simplifications. Note that the expression `R[1] + R[2]` part of the formula would be recognized by the

reduction rule for the aggregate function sum, but here we perform this substitution only for more than two terms.

Last, if we execute the routine with R = (50, 60, 70, 80, 90) and N = 5, the result will be

```
R_EQ = 1/(0 + 1/R[1] + 1/R[2] + 1/R[3]
           + 1/R[4] + 1/R[5])
| !(N = 2) and N > 2
```

If we simplify the formula by applying rule "0 + x ⇨ x" and the reduction rule for sum and simplify the path condition as well, we get

```
R_EQ = 1/SUM(1/R[i], 1, 5) | N > 2
```

The only difference between the expected specification shown in Eq. 1 and the deduced formulas is the upper bound of the sum (value 5 in formula vs. variable N in Eq. 1). This is a non-trivial remaining issue in our approach. Currently, the user can only speculate after several executions for N > 2, that the upper bound value of the resulting formula corresponds to the input value N.

Besides this limitation and if we are interested in cases with $N \geq 1$ only, we now have the expected specification given in Eq. 1.

B. Example: Quadratic Equation

Listing 2 gives program code that solves quadratic equation$AX^2 + BX + C = 0$. The program makes intensive use of *goto* statements, which results in a (typical) unstructured program code.

```
        Read(↑A ↑B ↑C)
        if B = 0 and C = 0 then goto L55 end
        if B ≠ 0 and C ≠ 0 then goto L50 end
        if A = 0 then goto L58 end
        if C ≠ 0 then goto L60 end
        XA := -B/A
        XB := 0
        goto L100
L50:    if A ≠ 0 then goto L60 end
        XA := -C/B
        XB := 0
        goto L100
L55:    if A = 0 then return end
L58:    Write(↓"TRIVIAL CASE. TWO OR MORE ZEROS")
        goto L111
L60:    Q := B*B - 4*A*C
        XX := -B/(2*A)
        if Q ≠ 0 then goto L80 end
L70:    XA := XX
        XB := XX
        goto L100
L80:    QA := Abs(↓Q)
        XS := Sqrt(↓QA)/(2*A)
        if Q < 0 then goto L110 end
L90:    XA := XX + XS
        XB := XX - XS
L100:   Write(↓"X1 = " ↓XA  ↓", X2 = " ↓XB)
        goto L111
L110:   XA := XS
        XB := -XS
        Write(↓XX ↓XA ↓XX ↓XB)
L111:   end
```

Listing 2. Pseudo code program to calculate the solutions of the equation $AX^2 + BX + C = 0$ (adapted from FORTRAN 77 code given in [14]).

Here, we illustrate the power of symbolic execution with respect to unstructured code and demonstrate how to answer question 1 ("*Which parameter impact on a particular computation?*"), because the program has no specification about given input and output data. The program reads the coefficients from a standard input (*Read*) and writes results to standard output (*Write*). In order to identify which variables are input data and which variables are output data (i.e. computation results), we analyze the program in an interactive way. When the *Execution Engine* comes around the *Read* instruction, the user is prompt to provide input for variables A, B and C.

We first analyze the program with A = 1, B = 5 and C = 6. In this case, we obtain two results *XA* and *XB* with the following symbolic formulas

```
XA = 0 - B/(2*A) + Sqrt(Abs(B*B - 4*A*C)) / (2*A)
XB = 0 - B/(2*A) - Sqrt(Abs(B*B - 4*A*C)) / (2*A)
```

and with the path condition for both formulas

```
!(B = 0 and C = 0) and !((B*B - 4*A* C) < 0)
```

One can immediately comprehend, that the solutions *XA* and *XB* are the common solution of a quadratic equation given in Eq. 2.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

The only differences are the *Abs* functions in the extracted formulas that protect the program from invocation of the *Sqrt* function with negative values.

Next, we analyze the program with $A = B = C = 1$ and get the following (already simplified with respect to *XX*) result

```
XA = Sqrt(Abs(B*B - 4*A*C)) / (2*A)
XB = -Sqrt(Abs(B*B - 4*A*C)) / (2*A)
XX = -B/(2*A)
```

and with the path condition `!(B = 0 and C = 0)`. In this case, we obtained formulas for complex roots. The formula for *XX* describes the *real part* whereas formulas for *XA* and *XB* describe the *imaginary part* of the equation for complex roots. Even the program has high cyclomatic complexity with eight if statements with corresponding *goto* statements, only two symbolic program executions yield the main solutions for a quadratic equation.

V. Related Work

Symbolic execution is applied in software engineering in various ways including *infeasible path detection* [15], *test input generation* [16] and *bug detection* [17]. Most symbolic analysis tools are motivated by software testing with the goal of exploring as many paths as possible [15]. Only a few approaches and tools are directly related to our approach facilitating maintenance and reengineering activities. Steward [18, 19] proposed a tool for rigorous program comprehension and error detection using symbolic execution and semantic analysis. In contrast to our approach, the resulting symbolic expressions are not the result but an intermediate representation

further processed by so called *expert parsers*. Expert parsers are able to detect certain properties such as physical equations provided that the program code is annotated by semantic declarations. The idea to use symbolic execution for deduction of the function computed by some elaborate or perhaps just hidden programs was already formulated by Fateman [20], however, this tool does not generate such functions. Instead, it uses symbolic execution to diagnose scientific computations. The DySy tool [13] uses dynamic symbolic execution to dynamically infer symbolic invariants from a program text. Inferred invariants are related to symbolic formulas deduced by our approach, but do not directly facilitate understanding which input parameter impact on a particular computation and how a program does achieve a particular result.

Huang [22] generates symbolic traces from instrumented programs. A symbolic trace is a linear listing of source statements and branch predicates that occur along an execution path in a program. Howden et al. [23] apply program slicing techniques to deduce functions from program code. Both symbolic traces as well as program slices are literally derived from the program text itself, without value substitutions as performed by symbolic execution and without simplifications by means of reductions rules.

VI. Conclusion

We proposed a novel approach and tool to extract specifications from source code by using the idea of symbolic execution. Though symbolic execution is applied in software engineering in various ways, this work is novel in exploiting benefits from symbolic execution to extract specifications from source code. The described tool is implemented as interpreter on an abstract syntax tree and covers common language concepts of procedural languages. Even we have applied our approach on single functions and small programs so far, we expect that it can facilitate maintenance and reengineering activities concerning legacy code in scientific and engineering domains.

References

[1] J. Pichler, "Extraction of documentation from Fortran 90 source code: an industrial experience," in Proceedings of 17th European Conference on Software Maintenance and Reengineering (CSMR'13), Genua, Italy, 2013, pp. 399–402.

[2] B. Bellay and H. Gall, "An evaluation of reverse engineering tool capabilities," Journal of Software Maintenance, vol. 10, no. 5, September 1998, pp. 305–331.

[3] H. M. Kienle and H. A. Müller, "The tools perspective on software reverse engineering: requirements, construction, and evaluation," Advances in Computers, vol. 79, Marvin V. Zelkowitz, Eds. Elsevier, 2010, pp. 189–290.

[4] H. Huang, "Business rule extraction from legacy code," in Proceedings of the 20th Conference on Computer Software and Applications (COMPSAC'96), Washington, DC, USA, 1996, pp. 162–167.

[5] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "A model driven reverse engineering framework for extracting business rules out of a Java application," in Proceedings of the 6th International Conference on Rules on the Web: Research and Applications, Berlin, Heidelberg, 2012, pp. 17–31.

[6] X. Wang, J. Sun, X. Yang, Z. He, and S. Maddineni, "Business rules extraction from large legacy systems," in Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR'04), Washington, DC, USA, 2004, p. 249–258.

[7] J. C. King, "Symbolic execution and program testing," Commun. ACM, vol. 19, no. 7, Jul. 1976, pp. 385–394.

[8] L. Clarke, "A system to generate test data and symbolically execute programs," IEEE Transactions on Software Engineering, vol. SE-2, no. 3, Sep. 1976, pp. 215–222.

[9] H. Dobler and G. Pomberger, Algorithmen und Datenstrukturen, Pearson Studium, April 2008.

[10] P. D. Coward, "Symbolic execution systems–a review," Software Engineering Journal, vol. 3, no. 6, Nov. 1988, pp. 229–239.

[11] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," in International Journal Software Tools & Technology Transfere, vol. 11, no. 4, Oct. 2009, pp. 339–353.

[12] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), New York, NY, USA: ACM, 2011, pp. 1066–1071.

[13] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: dynamic symbolic execution for invariant inference," in Proceedings of the 30th International Conference on Software Engineering (ICSE '08), New York, NY, USA: ACM, 2008, pp. 281–290.

[14] B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, 2nd ed., McGraw-Hill, 1978.

[15] W. Le and M. L. Soffa, "Marple: a demand-driven path-sensitive buffer overflow detector," in Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE'08), Nov. 2008, p. 272–282.

[16] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. (ESEC/FSE-13), New York, NY, USA: ACM, 2005, pp. 263–272.

[17] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09), 2009, pp. 225–236.

[18] M. E. M. Stewart, "An experiment in scientific program understanding," in Proceedings of the 15th IEEE International Conference on Automated Software Engineering, (ASE'00), Washington, DC, USA, 2000, pp. 281–284.

[19] M. E. M. Stewart, "Towards a tool for rigorous, automated code comprehension using symbolic execution and semantic analysis," in Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop (SEW'05), Washington, DC, USA, 2005, pp. 89–96.

[20] R. J. Fateman, "Symbolic execution and NaNs: diagnostic tools for tracking scientific computation," ACM SIGSAM Bulletin, vol. 33, nr 3, Sep. 1999, pp. 25–26.

[21] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05), vol. 40, no. 6. New York, NY, USA, Jun. 2005, pp. 213–223.

[22] J. C. Huang, "Instrumenting programs for Symbolic-Trace generation," Computer, vol. 13, no. 12, Dec. 1980, pp. 17–23.

[23] W. E. Howden and S. Pak, "The derivation of functional specifications from source code," in Proceedings of the 3rd Asia-Pacific Software Engineering Conference (APSEC'96), Washington, DC, USA: IEEE Computer Society, 1996, pp. 166–174.