Quantification of Software Changes through Probabilistic Symbolic Execution

Antonio Filieri
University of Stuttgart
Stuttgart, Germany
filieri@informatik.uni-stuttgart.de

Corina S. Păsăreanu
Carnegie Mellon Silicon Valley, NASA Ames
Moffet Field, CA, USA
corina.s.pasareanu@nasa.gov

Guowei Yang
Texas State University
San Marcos, TX, USA
gyang@txstate.edu

Abstract—**Characterizing software changes is fundamental for software maintenance. However existing techniques are imprecise leading to unnecessary maintenance efforts. We introduce a novel approach that computes a precise numeric characterization of program changes, which quantifies the likelihood of reaching target program events (e.g., assert violations or successful termination) and how that evolves with each program update, together with the percentage of inputs impacted by the change.**

This precise characterization leads to a natural ranking of different program changes based on their probability of execution and their impact on target events. The approach is based on model counting over the constraints collected with a symbolic execution of the program, and exploits the similarity between program versions to reduce cost and improve the quality of analysis results.

We implemented our approach in the Symbolic PathFinder tool and illustrate it on several Java case studies, including the evaluation of different program repairs, mutants used in testing, or incremental analysis after a change.

I. Introduction

Characterizing software changes is a fundamental component of software maintenance and differential analysis. Despite being widely used and computationally efficient, techniques that characterize *syntactic program changes* lack an insight on the changed program behaviors, which can possibly lead to unnecessary maintenance efforts. Recent promising techniques use program analysis to produce a behavioral characterization of program changes, see e.g., [1–4]. Nonetheless, such qualitative assessment provides only true-false answers, giving limited guidance on "how far" two different versions are from one another. We argue that a complementary *quantitative* representation for software changes is needed, particularly for programs required to operate under uncertain usage profiles, where the goal of maintenance is to improve the average quality of the program instead of its worst-case performance.

In this work, we propose to compute a precise *numeric* characterization of a program change by quantifying the likelihood of reaching program events of interest (e.g., successful termination, assertion violations, or the execution of a statement or a branch) and how that evolves in time, with each program version. Furthermore, our approach quantifies the percentage of inputs that are affected by each change. Such precise characterization of behavioral changes can be used to rank different program versions based on the execution probability of the changes and their impact on the probability of satisfying or failing desirable properties (considering also an uncertain usage profile, whenever available).

With this new quantitative approach we are able to state not only that the program has changed with a logical delta, as in the previous qualitative approaches, but we can also compute that delta affects say 30% of the program inputs, giving a clear, measurable indication for the effort necessary to re-test the program modifications or the relative risk due to the introduction of an undetected bug with the last change.

Furthermore, after fixing a bug, existing qualitative or testing-based techniques can only assess whether the new version is free of errors or not, which may be too restrictive for most realistic applications. Instead, with our approach we can quantify the probability of reaching an error in the old and new versions, expecting it to decrease with each new bug fix. In yet another scenario, consider the case of multiple candidate repairs for a given bug: our technique can be used to automatically rank them according to their probability of execution or the overall probability of failure.

The approach extends probabilistic symbolic execution [5, 6] to compute the symbolic constraints that characterize program paths in different program versions. Solution space quantification techniques [7, 8] over the collected constraints are used to precisely quantify the percentage of inputs leading to the occurrence of a target event that are affected by a change (approximate quantification with probabilistic precision guarantees has also been explored to cope with larger or nondeterministic programs [9, 10]). Furthermore, our approach exploits the fact that program versions are largely similar to reduce cost and improve the precision of analysis by storing and reusing partial analysis results from previous versions [11].

Besides sketching the potential applications we envision for our approach, we have implemented it in the Symbolic PathFinder tool [12] and performed an exploratory study on three possible applications: ranking of program repairs, incremental reliability analysis, and regression analysis of probabilistic programs. The preliminary results show a promising application scope for our technique.

II. Background on Symbolic Execution

Symbolic execution is program analysis technique that executes programs on symbolic rather than concrete inputs, and it computes the program effects as functions in terms of the symbolic inputs [13]. The behavior of a program P is determined by the values of its inputs and can be described by means of a *symbolic execution tree* where tree nodes are program states and tree edges are the program transitions as determined by the symbolic execution of program instructions.

```
1 void test(int x, int y) {
2   if(x>=0)
3     - ~~if(y>0)~~
3     + if(y>=0)
4       System.out.println("1");
5     else{
6       System.out.println("2");
7       assert x>y;
8     }
9   else
10    if(y>0)
11      System.out.println("3");
12    else
13      System.out.println("4");
14 }
```

Fig. 1. Example code change; the statement "if(y>0)" in the original program is changed to "if(y>=0)" in the new version.

The state s of a program is defined by the tuple (IP,V,PC) where IP represents the next instruction to be executed, V is a mapping from each program variable v to its symbolic value (i.e., a symbolic expression in terms of the symbolic inputs), and PC is a *path condition*. PC is a conjunction of constraints over the symbolic inputs that characterizes exactly those inputs that follow the path from the program's initial state to state s.

The current state s and the next instruction IP define the set of transitions from s. Without going into the details of every Java instruction, we informally define these transitions depending on the type of instruction pointed to by IP.

Assignment. The execution of an assignment to variable $v \in V$ leads to a new state where IP is incremented to point to the next instruction and V is updated to map v to its new symbolic value. PC does not change.

Branch. The execution of an *if-then-else* instruction on condition c introduces two new transitions. The first leads to a state s_1 where IP_1 points to the first instruction of the *then* block and the path condition is updated to $PC_1 = PC \wedge c$. The second leads to a state s_2 where IP_2 points to the first instruction of the *else* block and the path condition is updated to $PC_2 = PC \wedge \neg c$. If the path condition associated with a branch is not satisfiable, the new transition and state are not added to the tree.

Loop. A *while* loop is unrolled until its condition evaluates to false or a pre-specified exploration depth limit is reached. Analogous transformations are applied to other loop constructs.

The initial state of a program is $s_0 = (IP_0, V_0, PC_0)$, where IP_0 points to the first instruction of the main method, V_0 maps the arguments of main (if any) to fresh symbolic values, and $PC_0 =$ *true*. A program may also have one or more terminal states that represent conditions such as the successful termination of the program or an uncaught exception that aborts the program execution abruptly.

Our approach can be customized for any symbolic execution tool. We focus here on Symbolic PathFinder (SPF) [12] that analyzes Java bytecode programs.

III. Example

We illustrate our approach on the example in Figure 1. The program has two integer inputs: `x` and `y` (both ranging over -10 .. 9) and contains an assert violation (for the assertion at line 7). In the example, the statement `if(y>0)` (in red) in the original version of a program is changed to `if(y>=0)` (in blue) in a new, changed version of the program.

Figure 2 shows the symbolic execution tree of the original program, annotated with counters and probabilistic information computed by our quantitative analysis. The states in the tree contain information about the corresponding program statement, the number of inputs that reach that state, the percentage of inputs that reach that state given its parent has been reached and the percentage of inputs that follow the corresponding path in the program.

For example, the state $S3$, corresponding to the assert violation, will be reached by 110 input values out of the 400 in the domain, which represents 55% of the inputs that already reached the parent of the state in the tree. Further this represents 27.5% of the inputs that follow the program path leading from the tree root to the assert violation.

After the change described in Figure 1 is performed we obtain the annotated symbolic execution tree in Figure 3. Our tool identifies the portion of the tree that has been impacted by the change (in red) and re-computes the quantitative figures in an efficient way, only for the impacted portion of the tree (the values for the non-impacted nodes stay the same). In the second version, the assertion is still violated. However the number of inputs that reach the error is now 100, representing 50% of the inputs that reached the parent node, and 25% of the total number of inputs that follow the path leading to the error. Thus the percentage of inputs that lead to the error has decreased from 27.5% to 25%. Therefore, even if the change did not actually fix the bug, we can rank the two versions of the program based on the percentage of inputs that lead to the error, giving also a measure of the "likelihood" of executing the error (assuming all the inputs are equally likely). Furthermore, we can quantify the percentage of the inputs impacted by the change (e.g., 50% of the inputs lead from the root to the impacted node depicted with red in the figure), giving a measurable indication of the impact of the change on the behavior of the analyzed program. Finally, we can identify the branches that have been affected by the change by looking at the conditional probability; in this case the conditional probability for the branch in $S1$ changed, meaning the evaluated condition is statistically dependent on the change, while, for example, the one in $S2$, though predicate on `y`, is not because the change fall in a different scope.

Both the quantitative and the impact analysis are performed as described in the next section.

IV. The Approach

A. Probabilistic Analysis

We build upon previous work [6] where we defined a symbolic execution framework for computing the probability of successful termination (and alternatively the probability of failure) for a Java software component placed in a probabilistically uncertain environment. A failure can be any reachable error, such as a failed assertion or an uncaught exception, or reaching a specified unsafe state. For simplicity, we assume the satisfaction of target program properties to be characterized by the occurrence of a target event, but our work generalizes to bounded LTL properties [14].

To deal with loops, we run SPF using *bounded* symbolic execution, i.e., a bound is set for the exploration depths. The result of symbolic execution is then a finite set of paths, each with a path condition. Some of these paths lead to failure, some to success (termination without failure) and some lead to neither success nor failure (they were interrupted because of the bounded exploration) – the latter are called *grey* paths.

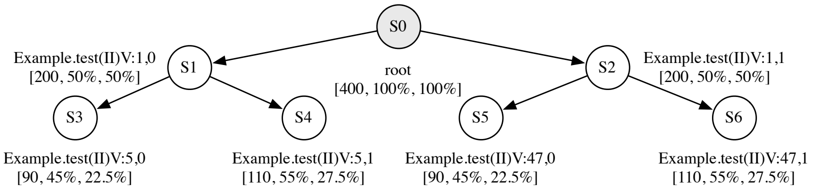

Fig. 2. Symbolic execution tree for the original program

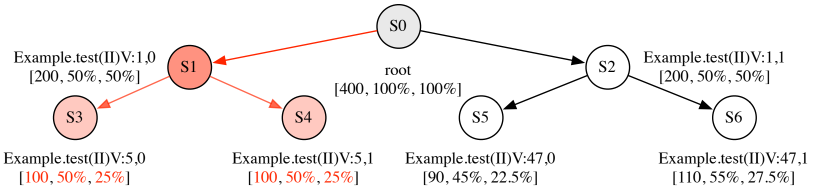

Fig. 3. Symbolic execution tree for the changed program

The path conditions produced by SPF consequently form three sets: PC^s=$\{PC^s_1,PC^s_2,...,PC^s_m\}$, PC^f=$\{PC^f_1,PC^f_2,...,PC^f_p\}$ and PC^g=$\{PC^g_1,PC^g_2,...,PC^g_q\}$, according to whether they lead to success, failure, or were truncated. Note that the path conditions are disjoint and cover the whole input domain. In other words, the three sets form a complete partition of the input domain [12, 13].

In its general form, the analysis we propose in this paper can take into account a probabilistic usage profile, i.e., a probabilistic distribution over the possible input variables resembling the expected user behavior [6]. In case no specific usage profile is provided, each input is considered equally likely. For the sake of simplicity, in this section we will focus on probabilistic analysis when no usage profile is available. To consider a usage profile, the equations in this section can be straightforwardly extended as in [6].

Given the output of SPF, the probability of success is defined as the probability of executing program P with an input satisfying any of the successful path conditions (recall the path conditions are disjoint):

$$Pr^s(P)=\sum_i Pr(PC^s_i) \tag{1}$$

The failure probability $Pr^f(P)$ and "grey" probability $Pr^g(P)$ have analogous definitions; it is straightforward to prove that $Pr^s(P)+Pr^f(P)+Pr^g(P)=1$. $Pr^g(P)$ can be used to quantify the *impact* of the execution bound on the quality of the analysis $(1-Pr^g(P))$.

In this paper we focus on sequential programs with integer inputs. In other work we provide treatment of multithreading [9], input data structures [6], and floating-point inputs [8]. We also proposed the systematic partial exploration of symbolic paths with statistical techniques in [10] to trade off accuracy of the analysis for scalability.

Quantification Procedure. We compute the probabilities of path conditions based on the quantification of their solution space (e.g., [5–7]). Since we are focusing on linear integer constraints, we can use LattE [7] to count the models satisfying them, but our work generalizes straightforwardly to other tools such as QCoral [8] (for arbitrary floating point constraints) and Korat [15] (for heap data structures; see [6] for details).

Given a finite integer domain D, model counting allows us to compute the number of elements of D that satisfy a given constraint c; we denote this number by $\sharp(c)$ (a finite non-negative integer). By definition [16], $Pr(c)$ is $\sharp(c)/\sharp(D)$ (where $\sharp(D)$ is the size of the domain implicitly assumed to be greater than zero).

The success probability (or failure or grey probability) can then be computed using model counting as follows:

$$Pr^s(P)=\sum_i Pr(PC^s_i)=\frac{\sum_i \sharp(PC^s_i)}{\sharp(D)} \tag{2}$$

where we implicitly assumed the domain D to be non-empty. The actual computation of $\sharp(\cdot)$ is optimized by a divide and conquer strategy and caching mechanism for intermediate results which allows for a significant reuse, as described in [6].

Conditional probability for a branch. Consider a branch in the symbolic execution tree that can be reached with a certain path condition $\bar{P}C$ and splits the control flow evaluating the condition b. The probability of satisfying b, given that the execution reached the conditional statement by satisfying $\bar{P}C$, can be formalized by the conditional probability:

$$Pr(b|\bar{P}C)=\frac{Pr(b\wedge\bar{P}C}{Pr(\bar{P}C)}=\frac{\sharp(b\wedge\bar{P}C)}{\sharp(\bar{P}C)} \tag{3}$$

Conditional probabilities can be used to identify the statistical dependence between a change in the program and the condition evaluate at a branch. If there is no dependency, the conditional probability of the branch remains the same, i.e., the contribution of the branch to the final success probability did not change since its condition is not affected by the change.

B. Incremental Analysis

The probabilistic symbolic analysis described above gives us a way for computing the probability of reaching target events (e.g., $Pr^s(P)$,$Pr^f(P)$) and the probability of executing different branches or statements. Computing these quantities, and then differencing them, for multiple program versions, gives us the precise numeric characterization of program changes that we are seeking. However, re-applying the probabilistic symbolic analysis to programs as they evolve may be impractical. We therefore aim to compute program differences that are utilized to make symbolic execution more efficient on the subsequent program version. The results generated by incremental analysis should be sound and complete, i.e., they must be the same as the results generated by regular symbolic analysis.

We build on previous work on Memoized Symbolic Execution (Memoise) [11], which leverages the similarities between successive problem instances that are analyzed by symbolic execution to reduce the total analysis cost by maintaining and updating the computations involved in a symbolic execution run. It reduces both the number of paths to explore by pruning the path exploration as well as the cost of constraint solving by re-using previously computed constraint solving results. Caching mechanisms are also used for the partial results of model counting procedures (adapting what described in [6]).

Memoise uses a trie [17, 18]—an efficient tree-based data structure—for a compact representation of the paths visited during a symbolic execution run. An initial run of Memoise performs standard symbolic execution as well as builds the trie on-the-fly and saves it on the disk for future re-use. Whenever a conditional instruction is symbolically executed a trie node is created. Specifically, in our approach, we store in each trie node bookkeeping information that maps each trie node to the corresponding condition in the code, i.e., method and the instruction offset of the symbolic conditional, the choice taken by the execution, as well as quantification information computed from probabilistic analysis. Figure 2 shows the trie for the original version of our running example.

Memoise enables efficient incremental analysis guided by the trie by only allowing the paths impacted by the change to be re-executed. Constraint solving is turned off for previously explored paths and the search is guided by the choices recorded in the trie. Moreover, the search is pruned for the paths that are deemed to be no longer of interest for the analysis.

A change impact analysis is used to identify the impacted trie nodes, which represent roots of sub-trees potentially changed by the execution of the change. Thus, only paths leading to the impacted trie nodes are selected be re-executed, and constraint solving is turned off for the portion of the path up to the impacted node. The control flow graph (CFG) of the program together with the trie are used to calculate the impacted trie nodes, and hence to guide symbolic execution to only execute paths with impacted trie nodes. Given a changed node in the CFG, we use backward reachability analysis to find the first symbolic conditional branch on each path from the changed node to the entry node in the CFG, and the trie nodes corresponding to the branch(s) are impacted.

Memoise monitors the symbolic execution of the program and whenever a conditional instruction is executed symbolically, it makes the corresponding traversal in the trie. Furthermore, Memoise turns off constraint solving for the portion of the path remain the same during re-execution, i.e., that corresponding to the path prefix leading to a impacted trie node in the trie. When encountering nodes that can not lead to any impacted trie nodes, the traversal backtracks and at the same time requests the symbolic execution to backtrack as well, thus "pruning" the search. When an impacted trie node is encountered, constraint solving is turned on. The part of the trie rooted at the impacted trie node is then built while new states are explored, using traditional symbolic execution. Constraint solving is turned off again when the traversal backtracks from a impacted trie node.

For example, in our running example, the change is made at line 3 of the program. Tracing the change towards the entry of the program in the CFG, we can find that the true branch of the symbolic conditional instruction at line 1 is the nearest symbolic branch leading to the change. We map this to the trie, and find the corresponding node $S1$, which represents the first choice, i.e., index 0; thus, $S1$ is the impacted node. Therefore, we select the trie path $S0 \rightarrow S1$ to guide the exploration; the execution corresponding to the other trie paths can be pruned; constraint solving is turned off for the execution corresponding to the selected path; it is turned on when $S1$ is encountered to rebuild the part rooted at $S1$. Figure 3 shows the updated example trie after the code change.

C. Potential application

In this section we sketch some of the main applications we envision for our technique. In the next section we will report on some preliminary experience.

Test suite evaluation and optimization. Quantitative program analysis naturally introduces the notions of *domain coverage* and *usage coverage* for a test suite. The former quantifies the fraction of the possible inputs actually covered by the test suite; the latter the fraction of the usage which is represented by the test suite, for a given usage profile. More metrics can be defined for other purposes. This quantitative information can be used to evaluate a test suite and to drive its improvement.

Evolutionary and search-based software optimization. These approaches aim at automatically modifying a program in order to improve its performance with respect to given metrics or to repair it, making it fit to specified requirements. Most techniques iteratively apply heuristics to generate variants of the original program; if such variants outperform the original one, they are used as basis for further improvement. The evaluation of a variant is usually based on predefined test suites, which may be partial or inadequate after the change. Our approach can be used besides or alternatively to testing to obtained global information useful not only to quantify the fitting to each requirement, but also possibly establishing a distance measure between alternative variants. Though generally slower than testing, the use of memoization and caching might make our technique "fast enough" for these applications, while providing richer information.

Improving fault localization techniques. Quantifying the impact of each part of the program on the occurrence of a failure can be a driver for new fault localization and primary cause analysis techniques. This approach can indeed provide more comprehensive information by counting the number of successful or failing tests in a predefined test suite.

Side-effects of local refactoring. When improving a part of the codebase, e.g., the implementation of a function, the global probability of failure should decrease. If this does not occur, the change is producing side effects on the program, which are ultimately reducing its probability of satisfying a requirement. The quantification of the global impact of a local change might provide a useful feedback to the developers while operating on a complex codebase.

Decision support for design choices. The precise quantification of the probability of satisfying relevant properties can be used to assess different design alternatives with comparable quality figures supporting design and coding choices.

Speeding up statistical symbolic execution. Memoization and caching for quantitative analysis can be beneficial to statistical symbolic execution approaches, where the symbolic execution tree is incrementally explored while collecting sample symbolic execution paths [10].

V. Experience

In this section we report on our preliminary experience on three potential applications of our technique: ranking of program repairs, incremental reliability analysis, and regression analysis of probabilistic programs. We have implemented our techniques in Symbolic PathFinder. We use Latte [7] for model counting over symbolic constraints.

A. *Ranking Program Repairs*

Consider the program in Figure 4. It is a code excerpt taken from TCAS, traffic collision avoidance systems [**tcas**]. This example is taken from [19] which demonstrates how the symbolic execution-based tool SemFix performs automatic repair.

```
1 int is_upward_preffered(int inhibit, int up_sep,
2     int down_sep) {
3         int bias;
4         int return_val;
5         if(inhibit>0) {
6            bias = down_sep; // bug
7            inhibit=1;
8         }
9         else {
10           bias = up_sep;
11           inhibit=0;
12        }
13        if (bias > down_sep) {
14           return_val=1;
15           assert(bias<=up_sep+100);
16        }
17        else {
18           return_val=0;
19           assert(inhibit*100+up_sep<=down_sep);
20        }
21        return return_val;
22 }
```

Fig. 4. Example code repair [19]

The intended behavior of the program is captured by the two assertions in the code, which are both violated for this example. The suspicious program statement that leads to error is on line 6 (see [19]) and suppose we have several available repairs for replacing line 6 with the following:

```
fix 1: bias=up_sep +300
fix 2: bias=up_sep +200
fix 3: bias=up_sep +100
```

These repairs can be created automatically with a tool like SemFix or are perhaps created manually by the developer. We can use our tool to rank the three bug fixes according to the likelihood of reaching error states. It turns out that for the 3 proposed fixes the assertions are still violated (SemFix only proposes fixes with respect to a set of test cases). In turn our tool reports the following "success" probabilities:

```
v0: 0.5033084513703282
v1: 0.5033344925054741
v2: 0.5033347658476457
v3: 0.5033350406475255
```

$v0$ is the original buggy version; $v1$, $v2$, and $v3$ represent the three versions using fixes 1, 2, and 3, respectively. Thus $v3 > v2 > v1 > v0$ according to the probability of program termination without failure. One can then argue that fix 3 is better than the other two fixes since it corrects more program behaviors, so it should be favored.

Furthermore, tools like SemFix address only one bug at a time, thus there is no guarantee a fix for a second bug will not compromise the effectiveness of the fix for the first bug. On the other hand, our technique computes the probability of a successful execution for all the admissible behaviors at the same time, thus encompassing the effects of all the bugs.

B. *Incremental Reliability Analysis*

To illustrate the effectiveness of our reuse strategies, we report on the incremental reliability analysis of a larger program, *MER* [20], which models a component of the flight software for JPL's Mars Exploration Rovers (MER); it consists of a resource: arbiter and two user components competing for five resources. MER has 4697 LOC (including the Polyglot framework). The software has an error (see [20]) and is driven by input test sequences. We analyze two versions: *MER (small)* for sequence length 8 and *MER (large)* for sequence length 20; the latter significantly stresses the tool due to the large amount of memory required to store its symbolic execution tree ($>8Gb$). For each of MER (small) and MER (large), we created another version by introducing a change to a randomly selected method in the program. We ran the initial run of Memoise to build the trie for the original version, and then re-use it for analyzing the changed version.

TABLE I. Incremental Reliability Analysis Results

Subjects	Techniques	Time (hh:mm:ss)	#States	#Solver Calls
MER (small)	regular	00:00:12	212	366
	incremental	00:00:20	94	144
MER (large)	regular	01:24:35	400,240	813,454
	incremental	00:00:34	6	0

Table I shows the results of applying both incremental analysis and regular analysis on the changed version (i.e., re-analyzing the program from scratch). For both regular analysis and incremental analysis, we report the time cost, the number of states explored, and the number of constraint solver calls involved in the analysis. We find that for MER (small), although incremental analysis reduced the number of states explored and the number of constraint solver calls, it cost 8 seconds more time due to the incremental analysis overhead; however, for MER (large), the savings in terms of time, number of states and constraint solving calls achieved by incremental analysis are significant because the change has little impact on the analysis that was done on the original program version.

C. *Regression Analysis for Probabilistic Programs*

Another application for our tool is regression analysis for probabilistic programs. Probabilistic programs are usual programs with two added constructs: (1) the ability to draw values at random from distributions, and (2) the ability to condition values of variables in a program via observations [21]. Models from diverse application areas such as computer vision, coding theory, cryptographic protocols, biology, and reliability analysis can be written as probabilistic programs. Our probabilistic framework allows a natural representation of such programs, where the probabilistic inference for probabilistic programs corresponds to computing probabilities of failure or success in our setting. For example, Figure 5 illustrates the encoding of the Bayesian network from [21].

The Bayesian network is composed by two nodes. The probabilistic choice within a node is formalized via the variables `BernoulliN`, each one simulating the flipping of a coin with rational bias. For example `Bernoulli5<10` has 9/100 probability of returning true (1..100 being the domain of `Bernoulli5`). The probability of successful execution of the first version (in Figure 5) is `0.37197435`, corresponding to

```
1  // all inputs range 1..100
2  void bayesN(int Bernoulli1, int Bernoulli2,
3  int Bernoulli3, int Bernoulli4, int Bernoulli5,
4  int Bernoulli6, int Bernoulli7, int Bernoulli8,
5  int Bernoulli9, int Bernoulli10) {
6       boolean i,d,s,l,g;
7       i = Bernoulli1<30? true : false;
8       d = Bernoulli2<40? true : false;
9       if(!i && !d)
10         g = Bernoulli3<70? true : false;
11      else if (!i && d)
12         g = Bernoulli4<95? true : false;
13      else if (i && !d)
14         g = Bernoulli5<10? true : false;
15      else
16         g = Bernoulli6<50? true : false;
17      assert(g); // instead of observe
18      if(!i)
19         s = Bernoulli7<5? true : false;
20      else
21         s = Bernoulli8<80? true : false;
22      if (!g)
23         l = Bernoulli9<10 ? true : false;
24      else
25         l = Bernoulli10<60 ? true : false;
26      assert(l); // probability of success gives the
27                 // conditional probability P(L|G)
28  }
```

Fig. 5. Example Bayesian network

the conditional probability of the event L conditioned to the occurrence of the event G. The symbolic execution tree is composed by 39 nodes and required 3 seconds for its analysis. The first version is then refined by modifying the first node of the Bayesian network by replacing the condition on line 14 with `Bernoulli5<80`. Reanalyzing the network regressively only required 13 nodes to be analyzed in **2** seconds, updating the computed probability to `0.44503405`.

VI. Related Work

Techniques that characterize syntactic program changes, e.g., *diff*, are imprecise leading to unnecessary maintenance efforts. More promising techniques produce a behavioral characterization of program changes, see e.g. [1, 2]. Behaviors are either abstracted through operational models (e.g., transition systems) or summarized through a set of logical formulae satisfied by the input-output relation (e.g., pre- and post- conditions). Checking the implication or the equivalence between the abstraction of different program versions provides a qualitative assessment of the preservation of desired behaviors or the elimination of undesired behaviors. These techniques provide only true-false answers. Recent work [3, 4] provides more informative but still only qualitative representation of program differences. In contrast, we have presented a complementary *quantitative* analysis of software changes that provides more information about the difference between program versions.

Our quantitative measures are different from *simulation distances* [22] which are real-valued functions between two high-level models (a specification and an implementation), computed using quantitative simulation games. Indeed, we focus on different versions of the same system, analyzing directly code (not high-level models), using probabilistic techniques.

We have presented the high-level ideas of this work in a one-page abstract at SNAPL'15 (snapl.org/2015, with no formal proceedings for the abstracts) with the goal of getting early feedback. In this paper we have developed the ideas further, we have also provided an implementation in Symbolic PathFinder and initial experimental results.

VII. Conclusions

We presented a technique for the quantification of software changes that uses symbolic execution and model counting to quantify precisely the difference between two program versions. We have implemented the analysis in Symbolic PathFinder, a symbolic execution tool for Java bytecode. We further used memoized symbolic execution and caching mechanisms for efficient reuse of results across versions. The computed quantitative information can be used in many maintenance and analysis scenarios, such as: evaluation of program repairs, refactoring (where we expect the probability of success to not decrease), probabilistic programming, search-based software engineering, fault localization, as well as providing new quantitative testing coverage criteria for evaluating and optimizing test suites. We plan to explore these directions further in future research as well as improving our implementation including more quantification procedures (floating point [8], strings, and data structures [6]), approximate counting [10], multithreading [9], and parallel symbolic execution to improve the scalability of our approach.

References

[1] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. "Differential symbolic execution". In: FSE '08. ACM, 2008, pp. 226–237.

[2] G. Yang, S. Person, N. Rungta, and S. Khurshid. "Directed Incremental Symbolic Execution". In: *ACM Transactions on Software Engineering and Methodology* 24.1 (2014), 3:1–3:42.

[3] N. Partush and E. Yahav. "Abstract Semantic Differencing for Numerical Programs". In: vol. LNCS 7935. SAS '13. Springer, 2013, pp. 238–258.

[4] N. Partush and E. Yahav. "Abstract semantic differencing via speculative correlation". In: OOPSLA '14. ACM, 2014, pp. 811–828.

[5] J. Geldenhuys, M. B. Dwyer, and W. Visser. "Probabilistic Symbolic Execution". In: ISSTA '12. ACM, 2012, pp. 166–176.

[6] A. Filieri, C. S. Păsăreanu, and W. Visser. "Reliability Analysis in Symbolic Pathfinder". In: ICSE '13. IEEE Press, 2013, pp. 622–631.

[7] J. A. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. "Effective lattice point counting in rational convex polytopes". In: *Journal of Symbolic Computation* 38.4 (Oct. 2004), pp. 1273–1302.

[8] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser. "Compositional Solution Space Quantification for Probabilistic Software Analysis". In: PLDI '14. ACM, 2014, pp. 123–132.

[9] K. Luckow, C. S. Păsăreanu, M. Dwyer, A. Filieri, and W. Visser. "Exact and Approximate Probabilistic Symbolic Execution for Nondeterministic Programs". In: ASE '14. ACM, 2014, pp. 575–586.

[10] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys. "Statistical Symbolic Execution with Informed Sampling". In: FSE '14. ACM, 2014, pp. 437–448.

[11] G. Yang, C. S. Păsăreanu, and S. Khurshid. "Memoized Symbolic Execution". In: ISSTA '12. Minneapolis, MN, USA: ACM, 2012, pp. 144–154.

[12] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis". In: *Automated Software Engineering* 20.3 (2013), pp. 391–425.

[13] J. C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394.

[14] P. Zuliani, A. Platzer, and E. Clarke. "Bayesian statistical model checking with application to Stateflow/Simulink verification". In: *Formal Methods in System Design* 43.2 (2013), pp. 338–367.

[15] C. Boyapati, S. Khurshid, and D. Marinov. "Korat: Automated Testing Based on Java Predicates". In: *SIGSOFT Software Engineering Notes* 27.4 (July 2002), pp. 123–133.

[16] W. Pestman. *Mathematical Statistics*. De Gruyter Textbook. De Gruyter, 2009.

[17] E. Fredkin. "Trie memory". In: *Communications of the ACM* 3 (9 1960), pp. 490–499.

[18] D. E. Willard. “New trie data structures which support very fast search operations”. In: *Journal of Computer and System Sciences* 28 (3 1984), pp. 379–394.

[19] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. “SemFix: Program Repair via Semantic Analysis”. In: ICSE ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 772–781.

[20] D. Balasubramanian, C. S. Psreanu, G. Karsai, and M. R. Lowry. “Polyglot: Systematic Analysis for Multiple Statechart Formalisms”. In: vol. LNCS 7795. TACAS ’13. Springer, 2013, pp. 523–529.

[21] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. “Probabilistic Programming”. In: FOSE 2014. ACM, 2014, pp. 167–181.

[22] P. Cerný, T. A. Henzinger, and A. Radhakrishna. “Simulation Distances”. In: vol. LNCS 6269. CONCUR ’10. Springer, 2010, pp. 253–268.