Automating Test-Suite Augmentation

Roderick Bloem*, Robert Könighofer*, Franz Röck*† and Michael Tautschnig‡
* Institute for Applied Information Processing and Communications, Graz University of Technology, A-8010 Graz, Austria
Email: {roderick.bloem, robert.koenighofer, franz.roeck}@iaik.tugraz.at
† NXP Semiconductors Austria GmbH, Gratkorn, A-8101 Gratkorn, Austria
Email: franz.roeck@nxp.com
‡ Queen Mary University of London, London E1 4NS, United Kingdom
Email: mt@eecs.qmul.ac.uk

***Abstract*—Test suites are hardly ever created from scratch. Hence, automatic test case generation methods should take advantage of existing tests to produce high-quality test suites more efficiently. We present an approach for automatic test suite augmentation addressing this challenge. It modifies existing test cases in such a way that full branch coverage in specified target functions is achieved. It is based on symbolic execution and model checking, and has been implemented as an extension to FShell, a test case generation tool for C programs. Finally, we present a case study where we apply our tool to augment a model-based test suite for real industrial code of a Java Card applet firewall, ultimately achieving 100% branch coverage fully automatically.**

***Keywords*-test suite augmentation; symbolic execution; model checking;**

I. Introduction

Testing is supposed to find bugs and increase the confidence in the quality of the product. While these objectives are hard to quantify, coverage goals are widely accepted means for estimating the progress and quality of the testing process. In practice, the effort for creating new tests often increases excessively the closer the test suite comes to satisfying the coverage goal. Automatic test case generation can help to find missing corner cases, but most methods create test cases from scratch and do not exploit already available ones.

In this paper, we consider test suite augmentation: Given a program and an initial test suite, we compute additional tests to maximize code coverage. We focus on branch coverage, but our approach can also be extended to other metrics. The idea is that some existing tests already get "close" to uncovered features, and can be altered to enter new terrain easily. This is potentially cheaper than computing new tests from scratch.

Besides manual test case generation, test suite augmentation is also useful in other scenarios. Test suites computed (automatically) with model-based testing [1], [2], [3] may cover the model well, but often fall short of achieving acceptable code coverage. The reason is that implementation details like null-pointer checks are omitted in the model, but also require proper testing. Also during the development or maintenance phase of a software product, code is changed and extended, but the regression test suite is usually not. This can make the coverage drop to poor rates, even if the initial test suite was of high quality. Automatic test suite augmentation, exploiting the available test cases, can fix this issue with acceptable effort.

This work was supported in part by the Austrian Research Promotion Agency (FFG) through project NewP@ss (835917).

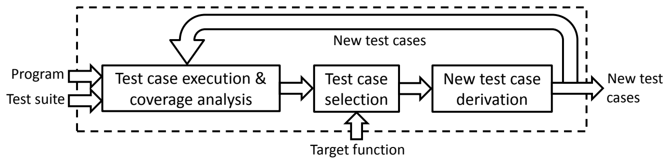

Figure 1. Workflow of our approach.

Figure 1 illustrates our approach. It takes as input a program, existing test cases, and a target function. It outputs additional test cases to achieve full branch coverage in the target function. First, we perform a *coverage analysis* of the existing tests by executing them on an instrumented version of the code. The next step is *test case selection*, where we select all test cases that pass a yet uncovered branch in the target function. When this set is empty, all branches in the target function are covered and our approach terminates. The last and central step is the *new test case derivation*. The path conditions of the selected test cases are investigated one by one with the goal of deriving a new test case that covers a previously uncovered branch. In the simplest case, this is possible by just negating the branching condition of the uncovered branching point and computing a satisfying assignment. If the formula is unsatisfiable, our approach backtracks along the execution path in order to search for a new path through the control flow graph to reach the desired branch. The search is guided by heuristics and parameters to configure them. Eventually (given enough time and resources), we will end up with a satisfiable path condition describing how we can reach the desired branch, because in the worst case the backtracking approach just iterates over all execution paths leading to the desired branch. Once a new test case is found, it is added to the test suite and the procedure is repeated.

We implemented our approach as an extension to FShell [4], building on its infrastructure for robust parsing of programs, the construction of formulas and path conditions, interfaces to solvers, etc. Our new approach consumes far less resources because it only analyzes one execution path at a time. If our backtracking does not succeed in reasonable time, we can fall back to FShell's default method for test case generation, which we also improve for our purpose by discarding irrelevant parts of the constructed formula. Finally, we evaluate our approach on an industrial implementation of a Java Card applet firewall, where we extend a test suite that has been derived from a model, and use this model as oracle.

In summary, this paper makes the following contributions:

1) We present efficient methods to augment an existing test suite such that it achieves branch coverage.
2) We describe an implementation of these methods, enhancing the FShell [4] test case generation tool.
3) We present a case study with real industrial code of a Java Card applet firewall implementation.

This paper is structured as follows. Section II discusses related work, Section III presents our approach in detail, Section III-F outlines our implementation in FShell, Section IV presents our case study and evaluation, and Section V concludes.

II. Background and Related Work

Generating meaningful input data is a fundamental problem in software testing. Over the last years, model checking tools made big strides and, therefore, received much attention in software verification and testing. A model checker verifies if a given program (model) satisfies a given specification. If not, a counterexample in form of concrete input values is produced to illustrate the specification violation. Tools such as CBMC [5], LLBMC [6], BLAST [7] and Java Pathfinder [8] are only a few of the available software model checking engines.

Model checkers can be used for test case generation [9] by generating so-called *trap properties* expressing that a desired situation can never occur. The model checker will then produce a counterexample demonstrating that it *can* actually occur. Using appropriate trap properties, a test suite achieving a desired coverage on the model can be derived automatically.

FShell [4] follows this idea using its own query language FQL for specifying which parts of the source code the user wants to cover. The underlying model checker CBMC [5] is used to build a formal representation of the program. Claiming then that the goals specified via FQL cannot be reached, counterexamples are constructed which satisfy the specified coverage goals. This approach is systematic and target oriented, but can be very resource demanding, because the model checkers operate on a formal representation of the *entire* program under test. Also, this approach does not benefit from existing test cases – a scenario which is more common than creating all test cases from scratch. Our test suite augmentation approach aims at eliminating these shortcomings.

Symbolic execution [10], is another test case generation approach related to our work. The program is executed using symbols (placeholders for concrete values) as inputs, tracking the symbolic values of program variables. When a branching point is reached, the execution forks. Along each path, a *path condition*, expressing the trace of this path, is computed. An SMT solver can then compute concrete inputs to activate a certain path. This approach is implemented in tools like KLEE [11]. The advantage of symbolic execution compared to model checking is that only one execution path is analyzed at a time. Our test suite augmentation method also works with path conditions, inheriting this advantage. On top of that, our methods are more target oriented (standard symbolic execution attempts to cover all paths) and exploit available test cases. The disadvantage of symbolic execution, as well as of our approach, is that the number of paths can explode. We address this issue with heuristics and parameters to guide the search.

Concolic execution is a mixture of symbolic and concrete execution, implemented in tools like DART [12], CUTE [13], CREST [14], and PathCrawler [15]. The program is executed with concrete inputs, while computing a symbolic path condition along the way. This path condition can then be (modified and) solved to calculate new inputs taking a different path. However, these tools do not take advantage of already existing test cases. Test suite augmentation using concolic execution is addressed in [16]. It explores alternative paths for existing test cases up to a certain iteration limit. In contrast, our method is more target oriented by analyzing the control flow graph in order to find feasible paths to a target branch. The work of [16] also presents a genetic approach based on selection of test cases according to a fitness function, and crossover operations to compute new test cases. A similar genetic approach is presented in [17]. Our test suite augmentation method is very different from these genetic approaches.

PathCrawler [15] aims to generate tests for all paths. It chooses a starting input randomly out of all possible inputs. It follows this path and calculates all inputs taking the same path. Those inputs are then excluded from the domain of possible inputs for new paths. In contrast, our approach augments an existing test suite by modifying the path condition of existing test cases such that a new branch is entered.

Among the existing tools and approaches, make-zesti [18], based on the KLEE symbolic execution engine, is one of the most similar tools to our work. It exploits an existing regression test suite to focus symbolic execution on interesting program features. As soon as the execution comes around "sensitive" operations like pointer dereferences, the tool explores additional paths around them. Besides the difference in the objective (testing sensitive operations vs. maximizing coverage), there is also a difference in the applied technique. make-zesti identifies points on the concrete execution path at which the execution can diverge, prioritizes them according to the distance to the sensitive operation, and uses them as starting point for symbolic execution. This is not guaranteed to hit the sensitive operation. In contrast, we analyze the control flow graph in order to find feasible paths to our target branch.

Another closely related tool is KATCH [19], a tool to increase the coverage on new patches committed to software repositories, also using KLEE. Like our work, KATCH also leverages existing test cases as a starting point for computing new ones. It finds the test case that is closest (according to some metric) to reaching the modified code. Then it applies symbolic execution with three heuristics to modify the test case in such a way that it reaches the target. Again, there are differences in the objective (covering code patches vs. maximizing coverage) and also in the methods and heuristics to reach them. For instance, KATCH only considers the one existing test case that is closest to reaching the target, while our approach iterates over all existing test cases to find one that can be modified to reach the target.

III. Test Case Generation

This section describes our test suite augmentation approach as outlined in Figure 1, as well as our implementation.

A. Test case execution and coverage analysis

In the beginning, we need to gather the coverage information for the existing test cases. For each test case, we track the sequence of branching points that are executed, and whether the respective condition evaluated to true or false. We achieve this by instrumenting the source code to store this information, and executing the test cases on the instrumented code.

B. Test case selection

After the function to cover got specified, we extract all test cases which enter the desired target function and pass at least one uncovered branch. This is done by searching our stored information for all conditions in the desired function which have no test case in either the true-branch or the false-branch.

Having the test cases which pass an uncovered branch, we can now apply different heuristics to generate new test cases. Our heuristics will be explained in the next sections. Every new test case gets executed immediately after discovery. The accompanying update of the coverage information possibly leads to a new set of test cases which can then be used by the heuristics again. Assuming that we can cover every branch, this set will become empty as soon as we covered the last uncovered branch. If the set is not empty but we cannot derive new test data, the still uncovered branch is potentially dead code and should be investigated by the test engineer.

C. Deriving New Test Cases: Simple Approach

A simple approach to derive a new test case is by walking through the program the same way an existing test case does, and gathering all conditions of the branching points into a path condition along the way: when the true-branch is taken by the test case, we take the condition as it is; when the false-branch is taken, we take the negated condition. When we reach the branching point with the yet uncovered branch, we do the opposite: if the test case takes the false-branch, we add the condition, otherwise the negated condition. No more conditions are added after the desired branch. The path condition so obtained evaluates to true whenever an input assignment takes exactly the same path as our initial test case, but the opposite turn at the desired branch. Hence, if the path condition is satisfiable, a new test case covering the branch is computed as a satisfying assignment of the path condition.

In a more formal way, we can define the procedure as follows: Let $\phi(path)$ be the path condition of the initial test case up until the first uncovered branch in the target function, excluding this branch condition. Let $\theta(branch)$ be this branch condition of the first uncovered branch in the target function. The initial test case satisfies $\phi(path) \wedge \theta(branch)$. If $\phi(path) \wedge \neg\theta(branch)$ is satisfiable, the new test case is computed as satisfying assignment to this formula.

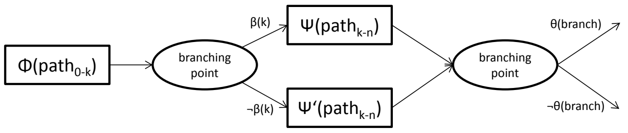

Figure 2. Illustration of the formulas when backtracking.

The advantage of this approach is that only a single path condition needs to be solved per uncovered branching point and existing test case. The path conditions are much simpler than the formulas representing the entire program, as used in the standard FShell approach. While this simple approach has the potential of hitting many previously uncovered branches at low costs, it may fail to cover certain branches due to dependencies of variables in the target condition.

D. Deriving New Test Cases: Backtracking

We use a backtracking approach to deal with unsatisfiable formulas occurring in the simple approach. While building the path condition, at every branching point we store the current path condition and current branching node on a stack. Intuitively, the stack contains yet unexplored possibilities for reaching the target branch. When we reach the desired branching point, but fail to compute a new test case with the simple approach, we analyze the control flow graph in order to find alternative paths to the target branch. This is done by traversing the control flow graph backwards, starting from the target branch, and marking all nodes that can reach the target branch in principle (without any semantic analysis). Next, we use the collected information for finding alternative paths to the target branch. We take the last branching point from the stack, take the other branch, and try to reach our desired branch, again pushing all branches that have not yet been investigated to our stack along the new way. Once a maximum path depth (defined by the user) is exceeded, the path condition becomes false, or we reach a node which cannot reach our desired branch according to the control flow graph, we stop the exploration and continue with the last branch from the stack. This is repeated until we either find a satisfiable path to cover the target branch, or exceed a maximum number of paths (defined by the user). Hence, our backtracking approach effectively implements a depth-first search of alternative paths to the target branch, restricted by reachability information from the control flow graph, and by user-defined parameters.

Figure 2 illustrates the backtracking approach in a more formal way. Let $\phi(path_{0-k})$ be the path condition of the test case up to the kth branch condition, but excluding this branch condition. Let $\beta(k)$ be the kth branch condition. Let $\psi(path_{k-n})$ be the path condition of the test case from the kth branch condition to the first uncovered branch in the target function, excluding this branch condition and the kth branch condition. Finally, let $\theta(branch)$ be this branch condition of the first uncovered branch in the target function. The current test case satisfies $\phi(path_{0-k}) \wedge \beta(k) \wedge \psi(path_{k-n}) \wedge \theta(branch)$. A new case is derived by solving $\phi(path_{0-k}) \wedge \neg\beta(k) \wedge \psi'(path_{k-n}) \wedge \neg\theta(branch)$, where $\psi'(path_{k-n})$ is a new path from the kth branch condition to the target branch. Our algorithm searches over different k and $\psi'(path_{k-n})$ until the formula becomes satisfiable.

The backtracking approach solves the problem of unsatisfiable formulas in the simple approach. The formulas only represent a single path, and are thus simple compared to the standard approach (building one monolithic formula) used by FShell and CBMC. When run without limits, and given enough time and memory, the approach will always find a test case to cover an uncovered branch if such a test case exists. The reason is that, in the worst case, the algorithm simply iterates over all paths to the uncovered branch. In theory, this number of paths may be infinite, so "enough time" may mean "infinitely long", which is not surprising because the underlying problem is undecidable. In practice, we use a bounded model checking approach in our implementation, which limits the number of execution paths to a finite number. This bound can be increased iteratively upon failure, until we run out of resources. Compared to the standard approach used by FShell and CBMC, our approach can handle much higher bounds in our experiments before running out of resources (especially memory). On the other hand, our approach may need many iterations, where FShell needs only one call to the model checker. The reason is that the amount of paths which need to be explored may explode, especially in programs with loops. In this sense, compared to FShell, we decompose the test case generation problem into several smaller problems, which are solved one after the other. Both approaches have their merits, and complement each other.

E. Reduced Formula Optimization

In our implementation, we can always fall back to test case generation with monolithic formulas using FShell. In this section, we describe an optimization of the standard FShell-approach in our setting. Inspired by ideas from program slicing [20], it builds the formula representing the program only for those parts that can reach the target branch. In contrast to static backward slicing, we only analyze the control flow graph and ignore data dependencies.

Similar to our backtracking approach, we mark in the control flow graph all preceding nodes of the desired branch as nodes which can possible reach our target branch. Then we restart the CBMC procedure to set up the formula for the program. However, when we reach a node which is not marked, we do not add the corresponding constraints to the formula and stop further exploration. In the end, the created formula contains only statements which could potentially be executed before reaching our desired target branch.

Despite this optimization, the formulas can become quite large, and the case generation approach resource demanding. Nevertheless, it can be a valuable alternative if the number of paths to search with our backtracking method explodes.

F. Implementation

The test case generation techniques presented in the previous sub-sections have been integrated into FShell [4], a test case generation tool which uses the bounded model checker CBMC [5] as a back-end. Using command-line parameters, the user can choose between the standard mode or one of our new methods. We also made part of the infrastructure required for test suite augmentation available from the command-line. For instance, it is possible to execute test cases right from the command line and integrate the gathered coverage information into the data file in which all the coverage information is collected. For generating test cases, the user specifies – using the available query language FQL – which function of the source code should be covered with branch coverage. In the automatic mode, FShell executes the new test case immediately after the data got generated and continues searching for new test cases until either every branch of the target function is covered or no new test case can be generated.

The files containing all the coverage information are stored in XML format and remain available after FShell is closed. There is one log-file for every test case, containing the actual path through the program. One additional file contains all passed conditions, and which test cases passed either the true- or the false-branch. Finally, one additional file stores the command line parameter for every test case and the according log-file name of their traces. Other tools can also use this information to calculate the coverage for different metrics, or present more detailed information to individual test cases.

IV. Evaluation and Case Study

In this section, we first evaluate our prototype implementation on two open source examples and then present a case study on real industrial code of a Java Card applet firewall.

A. Evaluation

Our first example is the *factor* implementation of the GNU coreutils[1] version 8.5. We had to make some small modifications, like inlining library functions such that our tool is able to handle and instrument the source code. The factoring function itself was not modified. It contains two nested loops where the loop conditions are input dependent.

Our initial test suite contains three test cases. The standard FShell implementation is not able to compute new tests that cover previously uncovered branches. As there are loops in the program, an unrolling bound has to be specified for CBMC. Low bounds are not sufficient to cover new branches. For higher bounds, the tool runs out of memory. In contrast, our backtracking approach is able to augment our existing test suite and derive the missing three test cases such that full branch coverage is achieved on this function. A single FQL query for generating a new test case takes between seven and fourteen seconds. Most of the time is spent for searching a new path to the desired branch. The time spent in the main SAT solver is only around half a second. This also points out

[1]http://www.gnu.org/software/coreutils/

Table I
TCAS RUNTIME RESULTS.

`alt_sep_test()`	Query CPU time	time in SAT solver
normal (FShell)	288 ms	68 ms
normal (backtracking)	232 ms	20 ms
main-loop (FShell)	27162 ms	25138 ms
main-loop (backtracking)	1190 ms	48 ms

the challenge of this example, which is the large number of possible paths within the a small amount of lines of code. We are aware that the comparison to FShell is not totally fair, because the goal of FShell is different from ours: FShell aims at test suite generation from the scratch, whereas our goal is test suite augmentation.

Our second example is the tcas[2] benchmark from the Siemens suite [21], implementing a traffic collision avoidance system for aircrafts in 180 lines of code. This program has 12 integer inputs. Our initial test suite contains only a single test. We compare our backtracking method with the standard FShell approach in achieving branch coverage on the top-level function `alt_sep_test()`. The results are summarized in Table I. FShell took 288 ms to generate a test suite from the scratch, while our approach required 232 ms to achieve full branch coverage. When we disregard the overhead and only look at the solving time, the relative difference is larger. The absolute solving time is small with either method, so we increase the complexity by adding a loop, and calling the top-level function only if the loop counter has a specific value. The computation time with FShell increases to 27 seconds, while our backtracking still succeeds in 1.2 seconds. We used a loop with 500 iterations in this experiment. With a higher iteration count, the difference becomes even greater. The reason is that the monolithic formula representing the program semantics grows very large due to the loop. Our approach works with path conditions, where the formula grows more moderately.

B. Case Study

To test our tool on real code from industry, we applied it to an applet firewall for a Java Card operating system. The Java Card operating system must be able to handle several (independent) applets. The firewall ensures that applets cannot arbitrarily access data belonging to other applets, but only in well defined cases, as defined in the JCRE specification [22]. It either grants access, or throws a security exception. In our implementation, the firewall is not implemented in one function, but is distributed between several functions.

We created an initial test suite using model-based testing techniques. The model formalizes the object access rules from Section 6.2.8 of the JCRE specification. Conceptually, it is very simple with only two states. However, the conditions on the edges, expressing when access is allowed or denied, are very complicated. We applied the method of [23] with the goal of achieving MCDC coverage [24] on these transition guards. This is done by generating equations expressing the MCDC criteria. An SMT solver then calculates satisfying assignments, which serve as abstract test cases. The test adapter then maps the values to concrete program variables in order to execute these test cases. It parses the input, sets the object flags and environment variables accordingly, calls the required firewall function and evaluates the obtained result by comparing it with the result obtained from the model.

Table II
COVERAGE RESULTS.

Branch Coverage	total	total (without unreachable)
basis test suite	58.11%	71.67%
augmented test suite	83.78%	**100%**

The so computed test suite consists of 211 test cases. Unfortunately, since the model is on a high abstraction level, it covers the source code only to a certain extend. We used gcov[3] to measure its branch coverage on the implementation. The results can be seen in Table II: only around 58 percent of the branches are covered. Some of the uncovered branches are security checks, which actually contain dead code branches under normal execution. There is no chance that these branches get covered during a normal execution. If we subtract those dead code branches from the result, we still cover only approximately 72 percent.

To improve this coverage, we applied our tool with backtracking on the firewall-related code of the Java Card operating system. We use a similar test adapter as for model-based testing. It uses the model as oracle also for our newly created test cases. Our tool generates 37 new test cases and identifies seven unreachable branches. As illustrated in Table II, the augmented test suites achieve 84 percent branch coverage (still including the dead code branches). All remaining uncovered branches in the firewall implementation are either unreachable by the design of the firewall due to security mechanisms or unreachable due to the design of the test adapter (e.g., certain pointers are never `null` because we always initialize them with valid values). After subtracting those unreachable branches from the result, the branch coverage is 100 percent. The only manual work required by the user is the inspection of branches which remain uncovered. Those uncovered branches are likely to be unreachable and have to be inspected. They can be marked as unreachable so that they are ignored in the next test suite augmentation session.

As this implementation of the applet firewall is already well tested and reviewed, it is not a surprise that no new bug was discovered. Nevertheless, the increased coverage gives a higher confidence in correctness.

As many of the initial test cases generated from the model are redundant with respect to branch coverage on the code, we also used a second test suite, which contains only one test case per firewall function. Hence, instead of starting with a test suite containing 211 test cases, we started with a test suite containing 7 test cases. Our test suite augmentation approach was able to complete the test suite with 41 additional test

[2]http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/

[3]http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

cases. The resulting test suite, which contains 48 test cases, achieves full branch coverage with respect to the reachable branches. The number of newly generated test cases is only sightly larger compared to the other case, where the generation started with the larger test suite.

Next, we compare our backtracking method with the standard FShell approach on one particular firewall function. While the standard FShell approach generates all test cases within a single FQL query, our method uses one FQL query per new test case. For the backtracking approach, we observed query times between two and seven seconds. The execution time roughly correlates with the number of steps which have to be made back along the path of the used test case to find a new path to the desired branch. The query CPU time for the backtracking algorithm was 3.8 sec on average if our initial test suite consisted of only a single test case. Four new test cases were created in this case. If the initial test suite contained already two test cases, the average time dropped to 2.7 sec per query (three new tests were created). For three existing test cases, two new ones could be computed within 2.6 sec on average. The standard FShell approach calculates a whole test suite within a single query. For the same function, this takes standard FShell 6 sec. Hence, with more than two existing test cases, our backtracking method is faster than standard FShell.

To evaluate our results, we also applied random testing to the firewall using the same test adapter. It required approximately 900 test cases with random inputs to achieve full branch coverage with respect to the reachable branches in the firewall functions. Compared to our test suite of 48 test cases, this difference in test suite size is quite significant.

V. Conclusion

We have presented an enhancement of FShell, making it capable of augmenting an existing test suite to achieve branch coverage on given target functions. Our approach generates new test cases by varying the paths of existing ones. This is potentially cheaper than computing new tests from scratch.

We presented and implemented three techniques. The first one is lightweight but incomplete, i.e., may fail to cover certain branches even if this is possible. The second one extends the former with backtracking to make it complete. Both techniques work with path conditions, which are simpler than formulas representing the entire program. The third technique is an optimization of the existing approach of FShell, restricting the formula construction to relevant parts of the program.

We evaluated our implementation on several smaller examples. Furthermore, we applied our tool on an industrial implementation of the firewall part of the Java Card operating system. Our tool augmented an existing test suite to achieve 100% branch coverage on the target functions. Our tool also identified dead code branches which are all due to security mechanisms in the code or restrictions by the test adapter.

In the future, we plan to enhance FShell by adding additional heuristics to trade execution time for memory consumption in a more flexible way. Instead of analyzing one execution path after the other, or all at the same time, we envision to operate on bundles of similar execution paths, thereby bridging the gap between symbolic execution and model checking even more. We will also research ways to enhance our heuristics with data flow analysis. Finally, additional coverage metrics can be added to the tool with low effort, as coverage collection and test case generation methods are already available.

References

[1] P. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *ICFEM'98*, 1998, pp. 46–54.

[2] A. J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *Softw. Test., Verif. Reliab.*, vol. 13, no. 1, pp. 25–53, 2003.

[3] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl, "Model-based test case generation for smart cards," *Electronic Notes in Theoretical Computer Science*, vol. 80, pp. 170–184, 2003.

[4] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "FShell: Systematic test case generation for dynamic analysis and measurement," in *CAV'08*, ser. LNCS, vol. 5123. Springer, 2008, pp. 209–213.

[5] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, ser. LNCS 2988. Springer, 2004, pp. 168–176.

[6] S. Falke, F. Merz, and C. Sinz, "The bounded model checker LLBMC," in *ASE'13*. IEEE, 2013, pp. 706–709.

[7] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.

[8] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA PathFinder," *STTT*, vol. 2, no. 4, pp. 366–381, 2000.

[9] G. Fraser, F. Wotawa, and P. Ammann, "Testing with model checkers: a survey," *Softw. Test., Verif. Reliab.*, vol. 19, no. 3, pp. 215–261, 2009.

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

[11] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08*. USENIX Association, 2008, pp. 209–224.

[12] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI'05*. ACM, 2005, pp. 213–223.

[13] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE'05*. ACM, 2005, pp. 263–272.

[14] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *ASE'08*. IEEE, 2008, pp. 443–446.

[15] N. Williams, B. Marre, P. Mouy, and M. Roger, "PathCrawler: Automatic generation of path tests by combining static and dynamic analysis," in *EDCC'05*, ser. LNCS 3463. Springer, 2005, pp. 281–292.

[16] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *FSE'10*. ACM, 2010, pp. 257–266.

[17] R. P. Pargas, M. J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Softw. Test., Verif. Reliab.*, vol. 9, no. 4, pp. 263–282, 1999.

[18] P. D. Marinescu and C. Cadar, "make test-zesti: A symbolic execution solution for improving regression testing," in *ICSE'12*. IEEE, 2012, pp. 716–726.

[19] ——, "KATCH: high-coverage testing of software patches," in *ESEC/FSE'13*. ACM, 2013, pp. 235–245.

[20] F. Tip, "A survey of program slicing techniques," *J. Prog. Lang.*, vol. 3, no. 3, 1995.

[21] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[22] Oracle, "Java Card 3 Platform Runtime Environment Specification, Classic Edition Version 3.0.4," 2011.

[23] R. Bloem, K. Greimel, R. Könighofer, and F. Röck, "Model-based MCDC testing of complex decisions for the Java Card applet firewall," in *VALID'13*, 2013, pp. 1–6.

[24] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/-decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.