SmartUnit: Empirical Evaluations for Automated Unit Testing of Embedded Software in Industry

Chengyu Zhang[1], Yichen Yan[1], Hanru Zhou[1], Yinbo Yao[2]
Ke Wu[2], Ting Su[3*] , Weikai Miao[1*], Geguang Pu[1*]
[1]School of Computer Science and Software Engineering, East China Normal University, China
[2]National Trusted Embedded Software Engineering Technology Research Center, China
[3]School of Computer Science and Engineering, Nanyang Technological University, Singapore
dale.chengyu.zhang@gmail.com,sei_yichen@outlook.com,hanruzh@gmail.com,snowingsea@gmail.com
bukawu@126.com,suting@ntu.edu.sg,wkmiao@sei.ecnu.edu.cn,ggpu@sei.ecnu.edu.cn

ABSTRACT

In this paper, we aim at the automated unit coverage-based testing for embedded software. To achieve the goal, by analyzing the industrial requirements and our previous work on automated unit testing tool CAUT, we rebuild a new tool, SmartUnit, to solve the engineering requirements that take place in our partner companies. SmartUnit is a dynamic symbolic execution implementation, which supports statement, branch, boundary value and MC/DC coverage.

SmartUnit has been used to test more than one million lines of code in real projects. For confidentiality motives, we select three in-house real projects for the empirical evaluations. We also carry out our evaluations on two open source database projects, SQLite and PostgreSQL, to test the scalability of our tool since the scale of the embedded software project is mostly not large, 5K-50K lines of code on average. From our experimental results, in general, more than 90% of functions in commercial embedded software achieve 100% statement, branch, MC/DC coverage, more than 80% of functions in SQLite achieve 100% MC/DC coverage, and more than 60% of functions in PostgreSQL achieve 100% MC/DC coverage. Moreover, SmartUnit is able to find the runtime exceptions at the unit testing level. We also have reported exceptions like *array index out of bounds* and *divided-by-zero* in SQLite. Furthermore, we analyze the reasons of low coverage in automated unit testing in our setting and give a survey on the situation of manual unit testing with respect to automated unit testing in industry.

CCS CONCEPTS

•Software and its engineering → Software testing and debugging; *Dynamic analysis; Empirical software validation;*

KEYWORDS

Dynamic Symbolic Execution, Automated Unit Testing, Embedded System

*Geguang Pu, Weikai Miao and Ting Su are the corresponding authors.

ICSE-SEIP '18, Gothenburg, Sweden

DOI: 10.1145/3183519.3183554

ACM Reference format:
Chengyu Zhang, Yichen Yan, Hanru Zhou, Yinbo Yao, Ke Wu, Ting Su, Weikai Miao, Geguang Pu. 2018. SmartUnit: Empirical Evaluations for Automated Unit Testing of Embedded Software in Industry. In *Proceedings of 40th International Conference on Software Engineering: Software Engineering in Practice Track, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE-SEIP '18),* 10 pages.
DOI: 10.1145/3183519.3183554

1 INTRODUCTION

Embedded software widely exists in various control systems, which is mostly specialized for the particular hardware it runs on and have different constraints, like time or memory. Manufacturers have broadly developed all sorts of embedded software in the electronics, *e.g.* cellphones, robots, digital TV *etc.* . Moreover, most of the equipments in industrial infrastructure extensively use embedded software, for instance, control systems in cars, trains, power plants, satellites, and so on. Thus, how to ensure the reliability and dependability of embedded software is an ongoing challenge for the safety-critical embedded systems.

Software testing is one of the most common ways to ensure the software quality. Many developers and researchers concentrate on how to improve the effectiveness and efficiency of the testing methods to achieve higher coverage and find more faults. Unit testing is an important step to ensure the software quality during the stage of software development [3, 28]. For example, 79% of Microsoft developers use unit testing in their daily work [41]. Meanwhile, unit testing is a mandatory task required in various international standards for different industrial systems, *e.g.* , IEC 61508, ISO26262, RTCA DO-178B/C *etc.* . For instance, IEC61508, which is intended to be designed as a basic functional safety standard applicable to all kinds of industry specifications, such as Safety Integrity Level (SIL), to provide a target to attain with respect to a system's development. If the software is up to level SIL 3/SIL4, both branch and MC/DC coverages have to be achieved to 100% during the unit testing stage. If not achieved, engineers are required to explore the software codes and find the reasons.

In general, the main objective in unit testing is twofold. One is to verify that the functionality is correct at the function level and the other is to ensure the function is fully tested and all possible branches and paths are taken. We call the former *functionality testing* and the latter *coverage-based testing*. Functionality testing is carried out in almost every software company as the basic quality

assurance means. Software engineers design the test cases manually in regards to the software design specification and then run the test cases to check the final results by specification or assertion. For coverage-based testing, software engineers may go through the codes and compute the conditions on branches/paths to obtain the test cases. During those activities, they usually utilize the commercial unit testing tools like Testbed[1], VectorCAST[2] *etc.* , to help them accelerate the task of test data design. Whatever functionality testing or coverage-based testing is involved, the work of test data design almost depends on manpower, which is a tedious job for software engineers.

In this paper, we aim at the unit coverage-based testing, and we believe that with the great advance achieved in the field of automated testing [1, 13, 32], especially in symbolic execution [7, 16, 31, 35] and decision procedure [4], we are capable of fully automatizing coverage-based testing in order to largely save manpower. To achieve this goal, firstly, we have elaborately investigated the real unit testing requirements from selected ten partner companies in China Mainland, covering main safety-critical fields like railway, aerospace, nuclear plant, and automobile. Secondly, by analyzing the collected requirements and based on our previous work on automated unit testing tool CAUT [33, 35, 36], we rebuild a new tool, *SmartUnit*, to meet the real engineering requirements that take place in those companies.

We observed that most of the companies bought kinds of commercial unit testing tools like Testbed or Tessy[3], which can support different chip platforms. They totally design test data by hand. In these ten companies, no one has used the automated testing tool in their production departments, but two of them have tried test data generation tools. The main reason for not adopting them is that existing commercial automated testing tools have achieved very low coverage but large test suites, since most of the existing tools are based on the random testing technology or simple branch analysis while ignoring the path analysis. We will discuss this more in section 5. For the tools from academia, like KLEE [7] or Otter [23] are far from mature in industry. In a word, it is quite surprising that NONE of the visited ten companies has adopted tools to help test data design, they still use the most traditional approach, manpower, to test the design for the safety-critical systems while the symbolic execution technique has already achieved great success in other fields like security and verification.

SmartUnit still follows the principle of symbolic execution approach [21] but has its novelty in the following points especially in the aspect of practice engineering.

(1) It is a dynamic symbolic execution (DSE) implementation. Based on the experience of developing CAUT [35], we elaborately design the execution engine of *SmartUnit* and make it robust enough since, in practice, an embedded project involving 20K lines of code on average cannot stop abnormally. We also design a new heuristic search strategy for speeding up automated unit testing, which supports statement, branch, boundary value, and MC/DC coverage.
(2) It can generate all the stubs automatically. One tedious work for unit testing is to design stubs to replace the existing function calls or global variables etc., the same in using commercial tools as well. SmartUnit makes it simpler. It also provides the options to leave the decisions for software engineers in case of adapting different application scenes.
(3) It is deployed as a private cloud-platform. Since the symbolic execution engine consumes computing resources heavily, it is designed as a private cloud-platform for internal use. Users only need to update the software project package by the web browser to the server, which will make the whole analysis automatically including stubs generation, symbolic execution, test data report generation *etc.* .
(4) It can be seamlessly integrated into the existent development environment, especially connecting to these commercial unit testing tools. Developers get used to the tools at hand, so one of the design philosophies for SmartUnit is to make the existent testing process as short as possible. To this end, SmartUnit can generate the test data input files for commercial tools like Testbed and Tessy. Once the test data suite is generated, it can be used by users directly in their unit testing tool at hand.

SmartUnit has been successfully applied in our four partner companies at the first stage from May to September 2017. For instance, our partners include *China Academy of Space Technology*, which is the main spacecraft production agency in China (like NASA in the United States); *CASCO Signal Ltd.*, which is the best railway signal corporation in China; and *Guangzhou Automobile Group*, which is one of biggest car manufacturers in China.

SmartUnit has been used to test over one million lines of code in real projects. For confidentiality motives, we select three in-house projects for the empirical evaluations, but we still cannot present the code example for the same reason. Thus, we carry out our evaluations on two open source database projects *SQLite* and *PostgreSQL*. We did not select open-source embedded software because we would like to test the scalability of our tool since the scale of the embedded software project is mostly not large, around 5K-50K lines of code. On the other hand, database projects are more complex than embedded software. From our experimental results, in general, more than 90% of functions in commercial embedded software achieve 100% statement, branch, MC/DC coverage, more than 80% of functions in SQLite achieve 100% MC/DC coverage, and more than 60% of functions in PostgreSQL achieve 100% MC/DC coverage. Moreover, SmartUnit has the ability to find the runtime exceptions at the unit testing level. We have also reported exceptions like *array index out of bounds* and *divided-by-zero* in SQLite.

The organization of the paper is as follows. Section 2 introduces the background of our tools and techniques, Section 3 shows the overview and the implementation details of our DSE-based C program unit test generation framework and its private cloud-platform, Sections 4 and 5 set up our evaluation and analyze the results, Section 6 discusses some related work, and Section 7 gives the conclusion.

[1] http://ldra.com/industrial-energy/products/ldra-testbed-tbvision/
[2] https://www.vectorcast.com/
[3] https://www.razorcat.com/en/product-tessy.html

2 BACKGROUND

2.1 Industry Situation

Unit testing is an important engineering activity to ensure the quality of software in industry, especially for the manufacturers of safety-critical systems, *e.g.* , the aerospace and railway signal control companies.

Although unit testing is a compulsory engineering activity requested by the standards, its application in industry is still suffering from low coverage and low efficiency due to the lack of automated tool. In most cases, test cases are first manually generated by engineers and then executed on the program code by some commercial third-party tools (*e.g.* Testbed, Tessy) to run the program code. Since manual test generation is time-consuming, the companies usually spend a lot of costs to employ test engineers or outsource in producing the test case. According to our industrial partners' experience, a trained test engineer can produce test case for 5-8 functions per day. One of our industrial partner spend over $10,000 per month for hiring a group of unit testing engineers while still suffers from the low efficiency and low fault detection. To tackle these challenges, a powerful tool that can automatically derive test cases of high coverage is highly desirable. Further, such a tool needs to be seamlessly integrated with mainstream third-party test execution tools. That is, the generated test data can be recognized by these tools to perform testing. It is inefficient and expensive, so they need efficient automatic tools to generate the test case for third-party tools.

2.2 Dynamic Symbolic Execution

Symbolic execution was first proposed by James C. King [21] in 1976. Due to the limited computing resource and SMT constraint solver, symbolic execution was not a practical technique in those years. Thanks to the recent computing resource improvement and a series of fantastic SMT solvers, such as Z3 [11], STP [8, 14], CVC4 [5], *etc.* , many symbolic execution engines have come into existence [9] (*e.g.* KLEE [7, 8], DART [16], CAUT [35] for C, JPF-SE [2] for Java). Researchers have also applied symbolic execution to software testing, including automatically generating test cases [22, 35].

Symbolic execution uses symbolic values as programs inputs to simulate the execution of programs. When dealing with a control-flow fork, symbolic execution engine collects the conditional expressions along the path as path constraint. When reaching the terminal of the program, SMT solver solves all the path constraint to get a result. The result is a test case that follows the path. The symbolic execution stops when all program paths are explored.

Dynamic **S**ymbolic **E**xecutive (DSE) is a variant symbolic execution, which was proposed in 2005 [16, 31], also called *Concolic Execution*. DSE uses concrete randomly generated values as input to execute the program while collecting path constraints during the execution. Then SMT solver solves a variant of the conjunction of these symbolic constraints to output a new input value. The new input value will be used to execute a new program path.

Figure 1 is an illustrative example to explain the symbolic execution. The code is a function named `checkSign` for checking signals, with its control-flow graph. If the input variable x is a positive number, the function returns value 1; if x is a negative number, the function returns value -1; otherwise 0;. The right side part of Figure 1 is the control-flow graph of the function.

```
1 int checkSign(int x){
2     if (x > 0)
3         return 1;
4     else if (x == 0)
5         return 0;
6     else
7         return -1;
8 }
```

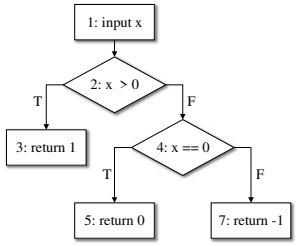

Figure 1: An example: *checkSign* function.

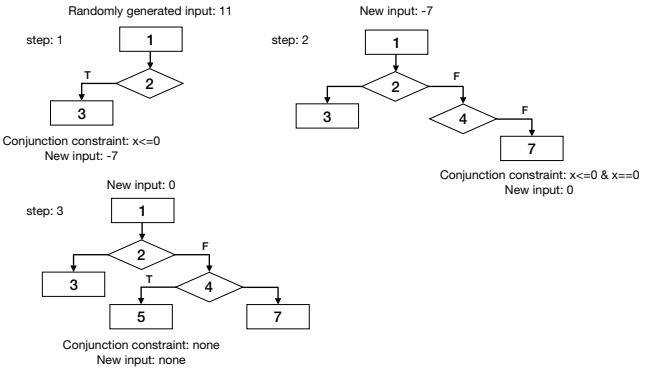

Figure 2: Process of dynamic symbolic execution.

Figure 2 describes the process of adopting the DSE. The goal is to cover line 7. DSE engine first randomly generates an input value, *e.g.* 11. Using this input value, the program reaches line 3. Then DSE engine negates the constraint collected from line 2 to solve a new input value, *e.g.* -7. Therefore the statement of line 7 is triggered. The DSE engine further negates the conjunction constraint collected from line 2 and line 4, generating another new input value, *e.g.* 0. Ultimately DSE engine can use value 0 as input to reach the statement of line 5.

Recently, a variety of DSE-based tools have been proposed [6–8, 16, 18, 24, 31, 38]. There are still some challenges for the DSE, *e.g.* exponential growth paths, symbolic pointer, guided execution, *etc.* . Section 3 will describe the implementation of our SmartUnit DSE-based engine and explain how these problems are solved.

2.3 Coverage Criteria

SmartUnit is a coverage-driven unit testing tool. One of its major goals is to generate the test suite towards a high coverage of code. In this subsection, we will introduce some commonly used coverage criteria in the industry.

2.3.1 Statement Coverage. Statement coverage requires all the statements in the program code under test be executed at least once by the test cases. Such coverage is easy to measure and the 100%

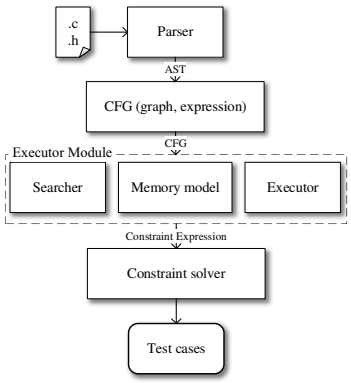

Figure 3: The architecture of SmartUnit.

statement coverage is also easy to achieve. Statement coverage is the most common used coverage criterion. For example, we can use it to detect the statements that are never executed. Since a large number of faults may not be detected by the criterion, it is usually not used alone.

2.3.2 Branch Coverage. Branch coverage is a stronger coverage criterion than statement coverage [45]. It needs to confirm all of the possible branches from each decision are executed at least once. The branch coverage is also easy to achieve the 100% coverage.

2.3.3 MC/DC coverage. MC/DC is the abbreviation of Modified Condition/Decision Coverage. It is a stronger coverage criterion than the branch coverage. In general, each decision is an atomic condition or combined with more than one atomic condition. When test cases satisfy each decision with value true and false, it obviously achieves branch coverage. However, MC/DC coverage further demands that test cases satisfying an atomic condition should affect decision independently with truth-value true and false, it is called Modified Condition/Decision Coverage. In practice, MC/DC-based unit test is usually difficult for the test engineer to write manually because of the complex logic in decision condition. But MC/DC coverage criterion is required in a variety of industrial standards. Therefore, test engineers always spend a huge amount of costs and time in designing the MC/DC test cases manually.

3 FRAMEWORK IMPLEMENTATION

3.1 Framework Architecture

Figure 3 shows the core framework of SmartUnit. The basic process of generating test cases in our approach is as follows:

SmartUnit accepts the `.c` and `.h` files as its input. To deal with macros and make sure some external symbols can be introduced into the source file, we use *libclang*[4] as preprocessor to generate processed .c file. Then libclang is also used to parse the processed file to generate the AST (abstract syntax tree).

We establish the CFG (Control Flow Graph) model based on the abstract syntax tree generated in the previous step. It consists of the control flow graphs generated from the proceeded files and the information of variables, expressions, functions *etc.* . Each node in the control flow graph represents a statement block in the source code. The sequential node contains exactly one incoming edge and one outgoing edge. The branch node contains one incoming edge and more than one outgoing edge and indicates the condition of the branch. The branch node usually represents if-else statement, while statement and switch statement *etc.* .

3.2 Executor Module

Executor module consists of the memory model, executor, and searcher. This part mainly processes the CFG model given by the previous steps.

3.2.1 Executor. Executor executes the statement expressions in the current node and drives the searcher to select next edge to explore. The executor does not really execute the C statements, it actually transforms the C statement into blocks, declarations or expressions structures that are stored inside SmartUnit. The executor updates information in the CFG model and adds the constraints to a path, after gathering them from the node statement. When the executor reaches the end node, it collects all the constraints on the path, and solves them by the constraint solver such as Z3.

3.2.2 Searcher and Search Strategy. To perform the search on control flow graph, we propose a new search strategy, named *flood-search policy*. Algorithm 1 describes the search algorithm. In our CFG model, each node in the model represents a basic statement block. The branch edges of the branch node record the branch condition and their truth values. If the input is `G(edges, nodes)` which is a CFG model. There are two execution state lists, open and close list. At the beginning, the algorithm starts with the initialized node into the open list, while the close list is empty. Then the algorithm executes the execution states in the open list in order. For each execution state, the algorithm will execute the shortest way from the current executed node to the exit node, and make sure the constraints in this path will be collected. In order to cover all branch edges in the graph, flood-search policy forks a copy of current execution state when it meets a fork, and adds the new execution state to the list corresponding to open or close. If all the succeeding nodes of a execution state have been visited, the execution state will be added to the close list. Comparing with other search algorithms such as breadth-first search (BFS) and depth-first search (DFS), flood-search is more suitable for dynamic symbolic execution, since flood-search in order to trigger the unvisited edges and nodes as quick as possible, meanwhile, BFS or DFS may fall in the loops.

3.2.3 Memory model. The memory model is the key module to track execution states and gather constraints, by simulating whole memory allocation. Basically, memory model stores all variables of basic types, such as `int`, `char`, including their names and values. Dealing with complex types, such as pointers, is a challenge for analysis tools. Our solution is below. For the array of basic types, in addition to memory space needed by the variable in the array, the total size of the array will also be stored, to perform plus/minus on pointers, and check if the pointer is out of memory bound. Furthermore, to deal with the member of `struct`, `union` or `enum`, the

[4] http://clang.llvm.org/doxygen/group__CINDEX.html

Algorithm 1: Flood-search for Control Flow Graph

Input: $G(edges, nodes)$: a control flow graph

```
1  open ← {State(Root(G))}, close ← {}
2  repeat
3      SearchShortestToExit(Pop(open));
4      if open.size = 0 then
5          Discharge();
6  until open.size = 0
7  Procedure SearchShortestToExit(State s)
8  if s is on the end node of the graph then
9      return
10 next_state ← Next(s);
11 if s has unvisited edge then
12     open ← open ∪ {s}
13 else if s has visited edge then
14     close ← close ∪ {s}
15 else
16     close ← close ∪ {s}
17     return
18 SearchShortestToExit(next_state);
19
20 Procedure Discharge()
21 repeat
22     open ← open ∪ {Next(Pop(close))}
23 until close.size = 0
```

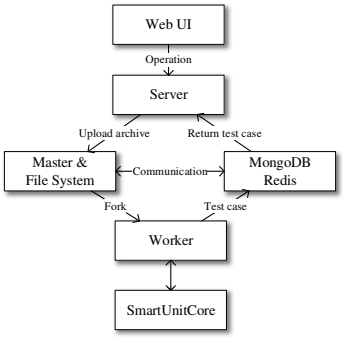

Figure 4: The workflow of the cloud-based platform.

total data structure and the start location in memory are recorded to locate the variable by start location and the memory offset.

3.3 Cloud-based Testing Service

The SmartUnit service contains all the other features, including web frontend UI, backend master process for handling web requests, worker process for performing analysis actions, database module for storing test results. The SmartUnit system architecture diagram is shown in Figure 4. The Web UI is designed to manage projects to be tested and get the results. The project under test needs to be uploaded to the server after archived, and will be passed to master process for further operations. The master process extracts the uploaded archive, creates a record in the database, and forks a worker process to call SmartUnit analysis engine (see Figure 3) to generate test cases for the project.

After SmartUnit finishes its analysis, worker process updates the status and saves generated test cases in the database. The status of Web UI is also updated, and then users could download test cases in specific formats, for example, `.tcf` format (for Testbed). When the master process receives a request for a specific test case, it will check if the test case is already prepared. If not, master process will generate a test case file according to test case data in the database and the required file format. When finished, the test case is returned to the Web UI, and could be downloaded.

3.4 Challenges and Solutions

3.4.1 Pointer. The symbolizing of the pointer variable is a challenge, because the pointer operations need to access the real value of the pointer and the variable that pointer points to. The real value of the pointer is hard to access as it is regarded as a symbolic value that means we must maintain a pointer array of pointer owner to execute a pointer operation, and the length of this array may be infinite because the pointer execution could be arbitrary. SmartUnit can support pointer operations, while a pointer memory must include the array memory of its owner which the pointer belongs to, the offset to address the position of the pointer, and some other marks such as null pointer mark. For the array of basic types, in addition to memory space needed by the variable in the array, the size of array will also be stored, to perform plus/minus operations on pointers, and check if pointer is out of memory bound.

3.4.2 void.* In most embedded C programs, (void*) is a special data type treated as a type that means nothing and used to transform data without data type. In Memory model, each pointer typed (void*) will be stored with its original type. SmartUnit implements this by maintaining a void memory type and saves the alias of the memory in it. When executor comes to the (void*), it will create a `voidmemory` type, and record its type information to the aliased memory of the voidmemory. It will update the aliased memory when executing to the assignment of void*, and look up to its alias memory rather than voidmemory itself, so that we can get the type and memory information of the void* type.

3.4.3 Complex data structure. Complex data structures like `struct`, `union` or `enum` type are hard to handle, for they do not have a fixed length. To deal with the members of the `struct`, `union` or `enum` type, the total data structure and the start location in memory are needed to locate the variable by start location and the memory offset. For a `struct` memory, the difficulty is to do operations related to the index. Thus, in addition to storing all variable declaration, the start and end memory location and relation between previous and next memory block are also needed to be stored.

4 EVALUATION SETUP

4.1 Research Questions

To evaluate our system, SmartUnit, we set up some research questions to guide our evaluation. The research questions are described as follows:

Table 1: Subjects of Evaluation Benchmark Repository.

Subject	# Files	# Functions	# LOC
aerospace software	8	54	3,769
automotive software	4	330	31,760
subway signal software	108	874	37,506
SQLite	2	2046	126,691
PostgreSQL	906	6105	279,809
Total	1028	9,409	479,535

- ***RQ1***: How about the performance of automated unit test generation framework, SmartUnit, on both commercial embedded software and open-source database software? We use statement coverage, branch coverage, and MC/DC as evaluation indicators.
- ***RQ2***: What factors make dynamic symbolic execution get low coverage?
- ***RQ3***: Can SmartUnit find the potential runtime exceptions in real-world software?
- ***RQ4***: What is the difference in terms of time, cost and quality between automatically generated test cases and manually written test cases?

We package the code under test and submit them to the SmartUnit cloud platform. SmartUnit can start dynamic symbolic execution automatically. After the execution, a series of packaged `.tcf` files (the test cases for Testbed) for the codes can be downloaded from the platform. We import the test cases into Testbed to get the coverage and detect the runtime error. Testbed will generate a detailed testing report after running the generated unit test cases. Statement, branch and MC/DC coverage and runtime errors (*e.g.* divided-by-zero, array index out of boundary) are provided by the report.

The statement, branch and MC/DC coverage of each function will be recorded as the performance indicator for *RQ1*. We turn our attention to the functions which get low coverage in the three coverage criteria above, to answer *RQ2*. The runtime errors will be classified into a several of categories, so that we can obtain some insights from them, in *RQ3*. It's impossible to manually write test cases for all of the benchmark, because of the large scale of the software in the benchmark. We will select some representative codes to answer *RQ4*.

4.2 Benchmark

This paper selects two kinds of C program benchmarks for SmartUnit: commercial embedded software and open-source database software. Table 1 gives the list of benchmarks for the evaluation. Due to the confidentiality agreement, we hide the commercial software names in this paper.

The commercial embedded software comes from aerospace, automotive, subway signal companies. Up to now, SmartUnit has already tested millions of code for a number of commercial embedded software. For example, in the aerospace company, SmartUnit has cumulatively tested more than 100,000 LOC. Over 70% functions have achieved more than 90% statement coverage. In this paper, to conduct an intensive study, we selected three benchmarks from different areas to ensure their diverse characteristics. All of them come from real-world industrial systems.

The open-source database software used in this paper are *SQLite*[5] and *PostgreSQL*[6]. Due to security demand of the commercial embedded software, we mainly use the open-source database software to explain. We chose SQLite because it is an embedded SQL database engine, usually used in embedded software systems. It is a good sample for us to find some insights when using SmartUnit on the embedded system, which has nearly 130,000 LOC. The PostgreSQL is a representative object-relation database system, usually used as the enterprise-class database. We chose PostgreSQL as a benchmark in order to evaluate performance and expandability of SmartUnit on the enterprise-class system.

For each subject, we put all of the `.c` and `.h` files into one folder to make it easier for Smartunit to get the dependent header files for the functions under test. We divided each subject into an independent `.zip` package so that we can calculate coverage respectively.

4.3 Evaluation Environment

SmartUnit was run on a virtual machine with three processorsTh 3GB memory, and CentOS 7.3 operating system. Testbed (version 8.2.0) was run on a virtual machine with two processors (2.70GHz Intel(R) Core(TM) i5-2500S CPU) 1GB memory and 32bit Microsoft Windows XP Professional Service Pack 3 operating system.

5 RESULTS AND ANALYSIS

RQ1: How about the performance of automated unit test generation framework SmartUnit on both commercial embedded software and open-source database software? For this question, we uploaded the benchmarks to the SmartUnit testing platform, made it run automatically, and got the coverage from the web UI. Table 2 shows the coverage information of the benchmarks. In Column *Subject*, the items represent the name of the programs in our benchmark. SmartUnit calculated coverage for each individual module based on its functionality, because the scale of PostgreSQL is quite large. Thus, we used PostgreSQL plus module names as the benchmark names in Table 2. The Column *#Test cases* represents the number of test cases generated by SmartUnit for the corresponding benchmark. The numbers in Statement Coverage, Branch Coverage, and MC/DC Coverage represent the number of functions in the corresponding range. The number of functions which achieve 100% coverage is highlighted in gray.

In general, more than 90% of functions in commercial embedded software achieve 100% statement, branch, MC/DC coverage, more than 80% of functions in SQLite achieve 100% statement, branch, MC/DC coverage, and more than 60% of functions in PostgreSQL achieve 100% statement, branch, MC/DC coverage. As Table 2 shows, most of the functions achieve 100% statement, branch, MC/DC coverage. The other coverage ranges have much less functions than 100% coverage range.

From the evaluation data, we can conclude that SmartUnit has a good performance on commercial embedded software and SQLite which is used in embedded systems. The performance on PostgreSQL is not as good as commercial embedded software and SQLite, but is also good enough. It indicates SmartUnit is more suitable for embedded software and also have a good enough performance on

[5]https://www.sqlite.org/
[6]https://www.postgresql.org/

Table 2: Performance of *SmartUnit* on statement , branch, and MC/DC coverage.

Subject	#Test cases	Statement Coverage (#Functions)					Branch Coverage (#Functions)					MC/DC Coverage (#Functions)				
		0%-10%	10%-50%	50%-90%	90%-100%	100%	0%-10%	10%-50%	50%-90%	90%-100%	100%	0%-10%	10%-50%	50%-90%	90%-100%	100%
aerospace software	712	1	4	5	4	40	1	4	5	3	40	3	-	-	-	51
automotive software	3134	1	3	9	2	315	1	6	8	-	315	3	3	1	-	323
subway signal software	14848	1	5	23	37	808	1	5	35	33	800	7	5	9	-	853
SQLite	6814	95	85	147	62	1657	98	111	144	66	1627	146	68	140	8	1684
PostgreSQL bootstrap	27	1	9	2	-	9	1	11	-	-	9	12	-	1	-	8
PostgreSQL catalog	1197	10	91	131	7	186	14	148	75	13	175	183	31	8	-	203
PostgreSQL initdb	312	-	2	3	6	61	-	4	10	11	49	2	-	9	-	61
PostgreSQL pg_dump	1608	16	27	60	25	378	20	36	60	38	352	37	22	32	-	415
PostgreSQL pg_resetxlog	65	-	-	2	1	8	-	-	2	1	8	-	1	-	-	10
PostgreSQL pg_rewind	245	4	1	2	4	53	4	1	2	5	52	4	-	4	-	56
PostgreSQL pg_upgrade	305	6	3	5	5	81	6	4	4	9	77	8	2	9	1	80
PostgreSQL pg_xlogdump	67	3	2	2	-	11	3	2	3	-	10	3	1	-	-	14
PostgreSQL pgtz	9112	587	595	704	68	2465	617	872	475	51	2404	1393	233	139	8	2646
PostgreSQL psql	1378	3	13	18	14	380	3	15	19	18	373	10	12	27	2	377
PostgreSQL scripts1	189	-	-	7	4	30	-	-	7	6	28	-	2	6	-	33

common software. However, there are some functions not achieving 100% of statement, branch, MC/DC coverage, we will discuss what factors make SmartUnit fail to get 100% coverage in RQ2.

RQ2: What factors make dynamic symbolic execution get low coverage? In Table 2, although SmartUnit has the high statement, branch, and MC/DC coverage, there are some function units has a low coverage (*e.g.* Th 0%-10%). We locate the low-coverage functions, inspect the source code and manually identify the reasons why SmartUnit gets low coverage in these functions. We categorized the main reasons as follows.

Environment variable and Environment function: In the benchmarks, there are a number of environment variables and environment functions in the code. For example, the current system time is an environment variable, and depends on the system. It is difficult to be symbolized. Environment functions are usually standard library calls, such as *sizeof()*. Listing 1 is an example from PostgreSQL. It is difficult to convert the condition in line 4 to a constraint, because symbolic execution engine cannot comprehend the semantics of the environment functions. Therefore, the coverage can not achieve 100% in this situation.

```
1  static void handle_sigint(SIGNAL_ARGS)
2  {
3          ...
4          if (PQcancel(cancelConn, errbuf, sizeof(errbuf)))
5          {
6              CancelRequested = true;
7              fprintf(stderr, _("Cancel request sent\n"));
8          }
9          else
10             fprintf(stderr, _("Could not send cancel request:
                   %s"), errbuf);
11         ...
12 }
```

Listing 1: An example of environment function.

Complex pointer operation: Although we have a solution to deal with pointer variables, it is still difficult to deal with various types of pointer operations. Listing 2 comes from SQLite. There is a complex pointer operation in line 4 which combines pointer variables and function pointers. Due to the limitation of the memory model, it cannot handle these complex operations. Execution will be terminated by this kind of statement and get a low coverage.

```
1  static void callFinaliser(sqlite3 *db, int offset){
2   ...
3   int (*x)(sqlite3_vtab *);
4   x = *(int (**)(sqlite3_vtab *))((char *)p->pModule + offset);
5   if( x ) x(p);
6   ...
7  }
```

Listing 2: An example of complex operation.

Limitation of SMT solver: In SmartUnit, Z3 is the main SMT solver used to solve constraints. Although Z3 is one of the best SMT solvers, it still has some limitations. Here is an example. *y != 0 && (((x-1) * y) % y) == 1* is a constraint collected by SmartUnit in the commercial software. Although it is a legal constraint and there is no divided by zero faults, the Z3 solver can not deal with it. The Z3 solver developer said that Z3 could not deal with nonlinear constraints, such as this constraint. It is the common reason for getting the low coverage. Thus the coverage of symbolic execution is sometimes affected by the SMT solver.

RQ3: Can SmartUnit find the potential runtime exception in real-world software? From RQ1, we found that SmartUnit has achieved high coverage on the real world software. During the execution, we observe some potential runtime exceptions in these software. Besides the factors discussed in RQ2, we found more than 5,000 test cases have witnessed runtime exceptions. Due to the time limit, we cannot check every test case manually, we randomly sampled some of test cases to analyze the runtime exceptions. Generally, we divided the runtime exceptions found by SmartUnit into three categories: array index out of bounds, fixed memory address and divided-by-zero.

Array index out of bounds: As introduced in Section 3, SmartUnit uses memory model to simulate the whole memory allocation. If there is an array index out of bounds, SmartUnit will throw a runtime exception. Listing 3 is an example from SQLite. Obviously, in this function, there is an out of bounds runtime exception in line 10, when $i < argc$. Although the caller of this function ensures $i <= argc$ in SQLite, potential runtime exceptions may occur if the other callers cannnot ensure $i <= argc$. There is even no preconditions is stated in the comment. It is quite serious if a programmer wants to call it, but doesn't know the precondition. We have found many of runtime exceptions in this category from all of the benchmarks.

```
/*
** Get the argument to an --option.  Throw an error and die if
** no argument is available.
*/
static char *cmdline_option_value(int argc, char **argv, int i){
    if (i == argc){
        utf8_printf(stderr, "%s: Error: missing argument to
              %s\n", argv[0], argv[argc - 1]);
        exit(1);
    }
    return argv[i];
}
```

Listing 3: An example of array index out of bounds.

Divided-by-zero: Divided-by-zero is also a common runtime exception type found in the benchmarks. It usually appears in numerical calculations. SmartUnit will generate boundary values to check if the program contains divided-by-zero runtime exceptions. We select a brief example from SQLite to discuss this runtime exception. Listing 4 is the function which has the potential runtime exception in line 19. When *nUsable == 4*, the expression *(nTotal - nMinLocal) % (nUsable - 4)* will throw a divided-by-zero runtime exception. we read the source code manually and found that there is no indications whether *nUsable* could be 4 or not. The comment of the function also does not state the precondition of *nUsable*. It's horrible if the programmer who calls this function in a new function and does not know the implicit prediction.

```
static void getLocalPayload(
  int nUsable,                  /* Usable bytes per page */
  u8 flags,                     /* Page flags */
  int nTotal,                   /* Total record (payload) size */
  int *pnLocal                  /* OUT: Bytes stored locally */
){
  int nLocal;
  int nMinLocal;
  int nMaxLocal;

  if( flags==0x0D ){            /* Table leaf node */
    nMinLocal = (nUsable - 12) * 32 / 255 - 23;
    nMaxLocal = nUsable - 35;
  }else{                        /* Index interior and leaf nodes */
    nMinLocal = (nUsable - 12) * 32 / 255 - 23;
    nMaxLocal = (nUsable - 12) * 64 / 255 - 23;
  }

  nLocal = nMinLocal + (nTotal - nMinLocal) % (nUsable - 4);
  if( nLocal>nMaxLocal ) nLocal = nMinLocal;
  *pnLocal = nLocal;
}
```

Listing 4: An example of divided by zero.

In summary, SmartUnit could find the potential runtime exceptions in real-world software. We categorized and analyzed the potential runtime exceptions. Most of these potential runtime exceptions exist due to there is no protection code for input values. At the same time, there is no clear precondition specification in the comments of these functions. Although there is protection code in the callers of these functions, we find these runtime exceptions are still causing real faults in real-world software. SmartUnit can effectively help developers find these potential exceptions before the release.

RQ4: What is the difference between automatically generated test cases and manually written test cases? In RQ4, we compare automatically generated test cases and manually written test cases in the following aspect: time, cost and quality.

Table 3: Test set generated time for each benckmarks.

Subject	# Functions	Time (s)	Average (s/func)
aerospace software	54	318	6
automotive software	330	329	1
subway signal software	874	2,476	3
SQLite	2046	13,482	6
PostgreSQL bootstrap	21	48	2
PostgreSQL catalog	425	1,350	3
PostgreSQL initdb	72	548	7
PostgreSQL pg_dump	506	3,428	7
PostgreSQL pg_resetxlog	11	71	6
PostgreSQL pg_rewind	64	352	5
PostgreSQL pg_upgrade	100	465	5
PostgreSQL pg_xlogdump	18	130	7
PostgreSQL pgtz	4419	10,478	2
PostgreSQL psql	428	1,676	4
PostgreSQL scripts1	41	311	7
Total	9,409	35,462	3.77

Time & Cost: Table 3 shows the test set generation time for each benchmark by SmartUnit. The column *Subject* lists all of the benchmark we used. Same as Table 2, we separate PostgreSQL into the individual modules, and name these modules with PostgreSQL plus the corresponding module name. The second column *# Functions* represents the number of functions in the benchmark. The column *Time* means the test set generation time for the corresponding benchmark. The column *Average* represents the average test set generated time for the corresponding benchmark per function, in other words, it means the average time of SmartUnit generates test set for a function. Additioanlly, at the last row, we give the number of total functions, total time of test generation, and the average time of test generation for all functions.

In Table 3, it is obvious that the average time ranges from 1s to 7s and total average time is 3.77s for all of the benchmarks. It means that SmartUnit spends nearly 4s to generate test cases for a function, and in the best situation, more than 90% of the functions can achieve 100% statement, branch, and MC/DC coverage. How about the test engineers? In our survey from the partner companies, a trained test engineer can product test case for 5-8 functions per day. Thus using automated unit test generation framework can cost much less time than employing test engineers.

Manually writing test cases takes a lot of efforts. According to our survey on the partner companies, a test engineer is able to write the test cases for 160 functions or so per month, nearly 2,000 functions per year. Meanwhile, SmartUnit is able to generate the test cases for 2,000 functions in two hours (as indicated by the data in Table 3) with high quality. Despite SmartUnit has cost 24 man-months to release, we believe SmartUnit is highly competitive especially when applied in large-scale systems.

Quality: From RQ1, we have the conclusion that more than 90% of functions in commercial embedded software can achieve 100% statement, branch and MC/DC coverage; more than 80% of functions in SQLite achieve 100% coverage, and more than 60% of functions in PostgreSQL achieve 100% coverage. According to our survey in the companies, the test engineers need to achieve 100% coverage for each function. If we use coverage as a quality indicator, automatically generated test cases have overwhelming superiority

on cost and time, although manually written test cases have higher coverage in some cases. Meanwhile, the automatically generated test cases could find runtime exception in time.

Discussion Traditional automated unit testing tools focus on the automated random testing, *e.g.* , test cases are randomly derived as the input to invoke the program code under test. Although the random test data generation algorithm is quite easy to be implemented, it is obviously that the expected coverage criteria (e.g., the branch and the statement coverage) cannot be guaranteed.

PEX is a famous unit testing tool that automatically generates test suites with high code coverage [38]. It is a Visual Studio add-in for testing .NET Framework applications. In practice, the PEX tool is adapted to testing the C# code. Similarly the IntelliTest[7] also automatically generates test data for the unit test of C#. These tools significantly improve the unit testing in bug detection and time efficiency. However, since these tools are designed as general solutions for unit testing, it cannot be directly applied in the domain of real-time control software testing. They fall short in supporting the unit testing of the control software since some unique features such as the high coverage request of particular criteria requested by the industrial standards have not been deliberately considered. The SmartUnit framework focuses on the request of current industry standards and offers a completely automated solution for the unit testing of the real-time software.

Although there is a plug-in of Testbed called LDRA TBrun[8] which can automatically generate driver program and test harness, SmartUnit is quite different from it. LDRA TBrun uses data dictionary to generate the test cases, it combines several values in the data dictionary, usually considering upper bound and lower bound of variables. This strategy is not able to deal with memory calculation, while SmartUnit can easily catch memory constraints by DSE. LDRA TBrun generates unit test suite based-on boundary value (*e.g.* , maximum value, minimum value, median value, *etc.*), while SmartUnit is a coverage-driven tool, it can satisfy statement, branch, MC/DC coverage criteria from industrial requirements. SmartUnit can avoid redundant test cases, while LDRA TBrun has many redundant test cases in its test set. LDRA TBrun cannot give the expected output values for each test case, but SmartUnit can give expected output values by executing test cases automatically after generation. SmartUnit not only generate the test cases for function parameters and global variables which were support in LDRA TBrun, it can also generate the test cases for instrumented function parameters and instrumented function return values. To sum up, LDRA TBrun is suitable for critical and robustness testing, while SmartUnit is suitable for coverage-driven testing.

During the evaluation, we found there are still some program language features cannot be executed automatically, for example, inline assembly and interrupts. For these cases, we rewrote the code and tried our best to keep the semantics. Moreover, floating point numbers and nonlinear constraints are also not fully supported due to the limitation of SMT solver. Fixed memory address is another challenging problem when dealing with pointer variables. In embedded systems, there are many pointer operations with fixed memory addresses. It is hard for the memory model to simulate a fixed memory address. For example, the operations like (*0X00000052) or (* (symbolic variable + 12)) will cause runtime exceptions. It usually gets *NULL* when referring a fixed memory address. Thus symbolic execution will throw runtime exceptions with fixed memory address operations.

6 RELATED WORK

This section discusses related work in two aspects: symbolic execution and automated unit test generation.

Symbolic execution is a classic software testing technique, and recently enhanced with the dynamic symbolic execution (also called concolic testing) technique [9, 10, 29]. There are several symbolic execution tool, such as Pex [38] for .NET, Java PathFinder [2], jCUTE [30] for Java, KLEE [7, 8], DART [16], CAUT [35], CUTE [30, 31], CREST [6] for C.

There is also much work on automated test generation. Some of them are based on Java. RANDOOP [26], EVOSUITE [12], AGITARONE[9] are usually used to generate test cases and evaluate on real-world software [1, 13, 32]. Some of them focus on unit test generation [15, 27, 42–44]. SmartUnit has a good performance on unit test generation, but also is evaluated on real-world software and used in practice. In the future, we will implement more advanced coverage criteria, *e.g.* , data-flow coverage [37], to further improve its fault detection ability; and extend SmartUnit to support other types of software (*e.g.* , mobile applications [34]) and scenarios (*e.g.* , requirement testing [25]).

In industry, *Microsoft* developed *Unit Meister* for parameterized unit tests [39] and *SAGE* for whitebox fuzzer testing [17]. *Fujitsu* tried to use symbolic execution to generate test [40], while *Samsung* used *CREST* and *KLEE* on mobile platform programs [19, 20]. The benchmarks they chose were all from their software. While We chose the benchmarks from a variety of embedded software systems and open-source software.

7 CONCLUSION

In this paper, we propose an automated unit coverage-based testing tool for embedded software called SmartUnit. It comprises of a dynamic symbolic execution engine, a unit test generator, and a private cloud-based service. It has been used in a series of real-world embedded software projects such as aerospace, airborne and ground-based systems. This tool has been developed and improved collaboratively with several top companies in China. The companies have used the SmartUnit in their daily testing process and improved their software reliability. We show a general pattern of how to use symbolic execution in practice with the example of SmartUnit.

The performance of SmartUnit is evaluated by testing with both commercial embedded software and open-source software. Besides, some runtime exceptions detected by our tool are collected and classified as guidance for potential users to avoid such runtime exceptions in software developing process. Challenges in using dynamic symbolic execution in the industrial environment have also been discussed.

In summary, although there are some challenges, it is possible to use dynamic symbolic execution technique on real-world software

[7] https://msdn.microsoft.com/en-us/library/dn823749.aspx
[8] http://ldra.com/industrial-energy/products/tbrun/
[9] http://www.agitar.com/solutions/products/agitarone.html

and get a high performance on coverage criteria. It is also practicable to build automated unit test generation tool as a cloud service to make unit testing easier to be adopted.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. Ting Su is partially supported by NSFC Projects No. 61572197 and No. 61632005. Geguang Pu is partially supported by MOST NKTSP Project 2015BAG19B02 and STCSM Project No. 16DZ1100600. Chengyu Zhang is partially supported by China HGJ Project (No. 2017ZX01038102-002).

REFERENCES

[1] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Jānis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 263–272.

[2] Saswat Anand, Corina Păsăreanu, and Willem Visser. 2007. JPF–SE: A symbolic execution extension to java pathfinder. *Tools and Algorithms for the Construction and Analysis of Systems* (2007), 134–138.

[3] IEEE Standards Association et al. 1990. Standard glossary of software engineering terminology. *IEEE Std* (1990), 610–12.

[4] Clark Barrett. 2013. fiDecision Procedures: An Algorithmic Point of View,fi by Daniel Kroening and Ofer Strichman, Springer-Verlag, 2008. *Journal of Automated Reasoning* 51, 4 (2013), 453–456.

[5] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

[6] Jacob Burnim and Koushik Sen. 2008. Heuristics for scalable dynamic test generation. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE, 443–446.

[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[8] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 10.

[9] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 1066–1071.

[10] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

[12] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 416–419.

[13] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8.

[14] Vijay Ganesh and David L Dill. 2007. A decision procedure for bit-vectors and arrays. In *CAV*, Vol. 4590. Springer, 519–531.

[15] Pranav Garg, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. 2013. Feedback-directed unit test generation for C/C++ using concolic execution. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 132–141.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM Sigplan Notices*, Vol. 40. ACM, 213–223.

[17] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Queue* 10, 1 (2012), 20.

[18] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing.. In *NDSS*, Vol. 8. 151–166.

[19] Yunho Kim, Moonzoo Kim, and Yoonkyu Jang. 2011. Concolic testing on embedded software-case studies on mobile platform programs. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE) Industrial Track*, Vol. 29. 30.

[20] Yunho Kim, Moonzoo Kim, Young Joo Kim, and Yoonkyu Jang. 2012. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 1143–1152.

[21] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[22] Guodong Li, Indradeep Ghosh, and Sreeranga Rajan. 2011. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In *Computer Aided Verification*. Springer, 609–615.

[23] Kin-Keung Ma, Khoo Yit Phang, Jeffrey Foster, and Michael Hicks. 2011. Directed symbolic execution. *Static Analysis* (2011), 95–111.

[24] Rupak Majumdar and Ru-Gang Xu. 2009. Reducing Test Inputs Using Information Partitions.. In *CAV*, Vol. 9. Springer, 555–569.

[25] Weikai Miao, Geguang Pu, Yinbo Yao, Ting Su, Danzhu Bao, Yang Liu, Shuohao Chen, and Kunpeng Xiong. 2016. Automated Requirements Validation for ATP Software via Specification Review and Testing. In *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings*. 26–40.

[26] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 815–816.

[27] Brian Robinson, Michael D Ernst, Jeff H Perkins, Vinay Augustine, and Nuo Li. 2011. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 23–32.

[28] Per Runeson. 2006. A survey of unit testing practices. *IEEE software* 23, 4 (2006), 22–29.

[29] Koushik Sen. 2006. *Scalable automated methods for dynamic program analysis*. Technical Report.

[30] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, Vol. 6. Springer, 419–423.

[31] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 263–272.

[32] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 201–211.

[33] Ting Su, Zhoulai Fu, Geguang Pu, Jifeng He, and Zhendong Su. 2015. Combining symbolic execution and model checking for data flow testing. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 654–665.

[34] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 245–256.

[35] Ting Su, Geguang Pu, Bin Fang, Jifeng He, Jun Yan, Siyuan Jiang, and Jianjun Zhao. 2014. Automated coverage-driven test data generation using dynamic symbolic execution. In *Software Security and Reliability, 2014 Eighth International Conference on*. IEEE, 98–107.

[36] Ting Su, Geguang Pu, Weikai Miao, Jifeng He, and Zhendong Su. 2016. Automated coverage-driven testing: combining symbolic execution and model checking. *SCIENCE CHINA Information Sciences* 59, 9 (2016), 98101.

[37] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. 2017. A Survey on Data-Flow Testing. *ACM Comput. Surv.* 50, 1, Article 5 (March 2017), 35 pages.

[38] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–white box test generation for. net. *Tests and Proofs* (2008), 134–153.

[39] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests with unit meister. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 241–244.

[40] Susumu Tokumoto, Tadahiro Uehara, Kazuki Munakata, Haruyuki Ishida, Toru Eguchi, and Masafumi Baba. 2012. Enhancing symbolic execution to test the compatibility of re-engineered industrial software. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, Vol. 1. IEEE, 314–317.

[41] Gina Venolia, Robert DeLine, and Thomas LaToza. 2005. Software development at microsoft observed. *Microsoft Research, TR* (2005).

[42] Tao Xie and David Notkin. 2003. Tool-assisted unit test selection based on operational violations. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. IEEE, 40–48.

[43] Sai Zhang, David Saff, Yingyi Bu, and Michael D Ernst. 2011. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 353–363.

[44] Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. 2010. Random unit-test generation with MUT-aware sequence recommendation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 293–296.

[45] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *Acm computing surveys (csur)* 29, 4 (1997), 366–427.