Exhaustive Test-case Generation using Symbolic Execution

● Tadahiro Uehara

Software testing has been one of the major challenges in the development of software for enterprise systems because it accounts for 30% to 50% of the total development cost required. Meanwhile, such testing has become increasingly important, because the upcoming paradigm for information and communications technology (ICT) system development, such as test-driven development and continuous integration, is directed at automated testing. Over a period, Fujitsu Laboratories has pursued R&D of software testing, mainly focusing on the test-case generation, which has a major impact on ensuring software quality and minimizing the development cost required. Such test-case generation has been successfully realized in the test function of the FUJITSU Software Interdevelop Designer—Fujitsu's business application development platform. We recognized early on the great potential of symbolic execution, a method that has become a popular academic research topic today. This paper presents an exhaustive test-case generation technology that utilizes symbolic execution. It also describes three issues to be overcome for its practical application, aiming to improve the efficiency of program unit testing and regression tests for version upgrades. This is followed by accounts of the approaches adopted to overcome these issues. This paper also introduces some cases in which we evaluated its application to working software assets.

1. Introduction

Software testing has long been recognized as one of the important themes in software engineering, and many technological innovations have been achieved in this area both in industry and in academia. However, software testing accounts for 30% to 50% of software development man-hours in current enterprise systems, and thus remains a major challenge. Further, software testing is becoming increasingly important as software development paradigms predicated on the automation of testing, such as test-driven development, which consists in creating tests before program development, and continuous integration, which promotes early detection and fixing of bugs through the daily and continuous execution of automated testing, are becoming mainstream.

Table 1 lists the test automation technologies currently in practical use in the testing phase of business application development. The testing phases are listed along the vertical axis, and the tasks at each phase along the horizontal axis.[1)] The tasks for which automation is making progress are test execution and test management. For test execution, unit testing frameworks, such as JUnit, have been developed for each development language. Likewise for integration tests and system tests, capture & replay type tools that, once testing procedures have been recorded, can automatically replay them later, are available as commercial tools and as open-source software, providing essential tools for modern development practices.

On the other hand, in the area of test analysis, test design, and test implementation, in other words test-case and test-data generation, automation has not progressed. There are combination generation tools based on the orthogonal array[2)] and the All-pairs testing method,[2)] but determining which test conditions need to be extracted, which directly affects test quality, is done by humans. Fujitsu Laboratories is carrying out research and development on how to extract these test conditions to improve and standardize test quality.

Table 1
Test automation technologies in testing phase of business application development.

Test level	Test analysis	Test design	Test implementation	Test execution	Test management
	Test-case generation		Test-data generation		
Unit test		Coverage measurement tool		Unit testing tool	Vendor tools, Jenkins plugin, etc.
Integration test		State transition testing tool, integration testing tool		Capture & replay tool	
System test	Research target			Performance testing tool, Security testing tool	
Acceptance test					

This paper introduces the various technologies related to test-case generation that the authors have conducted research on so far. First, it introduces symbolic execution, which is at the core of test technology. Next, it describes technologies for generating test cases for unit testing, and introduces technologies for application to regression testing during program revisions. Last, it discusses future initiatives and prospects.

2. Test-case generation using symbolic execution

Symbolic execution is technology for the exhaustive extraction of paths executable by software. Research on this technology has been conducted since the 1970s,[3] yielding various achievements in the academic field, but owing to high computational cost, no progress had been made in terms of practical use. However, from the 2000s, thanks to rapid advances in satisfiability problem (SAT)/satisfiability modulo theories problem (SMT) techniques[4] for quickly solving the constraint satisfaction problem (problem of finding a solution that satisfies given formulas), analysis using symbolic execution for programs of a practical scale became feasible in a reasonable computational time period. At present, the generation of test cases using symbolic execution is a major technology area in software engineering being worked on by many researchers.[5] Early on, Fujitsu Laboratories became interested in the possibility of applying symbolic execution testing to various tests, and has been conducting research in this area.[6] Research on one of these test areas, the generation of test cases for unit testing, has been carried to a practical application level, resulting in provision as a test function for the FUJITSU Software Interdevelop Designer, Fujitsu's development environment for business applications.[7]

The principle of symbolic execution is explained below. Symbolic execution is an execution method that treats inputs as symbolic values without concrete values. When determining the conditional branches including symbols, both branches, namely true and false, are executed respectively, recording conditional expressions for symbols. At that time, taking into consideration the conditional expressions (path conditions) recorded up to that point, whether each case is possible is judged using an SMT solver, and the processing continues only the branch whose case is judged to be possible. By repeating this procedure, it is possible to extract all the executable paths of the program.

Application of symbolic execution to test-case generation consists in treating the extracted paths as test cases. If the given conditional expressions are possible, the SMT solver outputs sample concrete values for the symbols. The executable test cases are obtained by utilizing this value as input data for symbolic execution.

Let us illustrate this by using **Figure 1** as an example. Taking symbol Sym*x* as argument *x*, there is a branch evaluating the symbol variable on the line 2 and line 4 of the program to be tested, so the path conditions are as follows.

1) (Sym*x*=123)&(Sym*x*+1<0)
2) (Sym*x*=123)& not(Sym*x*+1<0)
3) not(Sym*x*=123)&(Sym*x*<0)
4) not(Sym*x*=123)& not(Sym*x*<0)

However, in the case of 1), there is no value for Sym*x* that satisfies this path condition, and it is

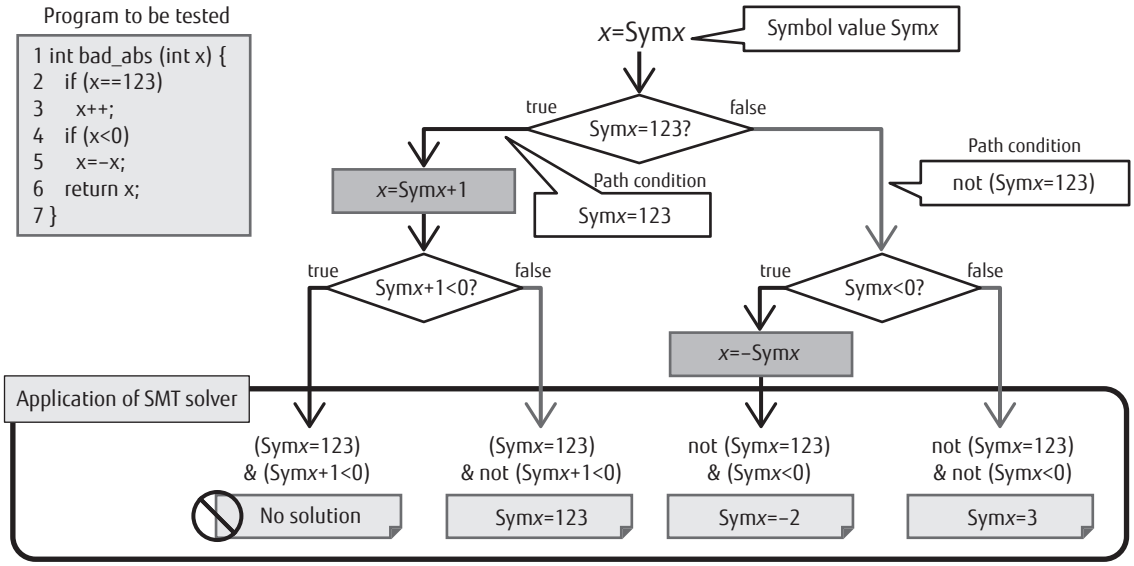

Figure 1
Mechanism of symbolic execution.

therefore judged to be a non-executable path and no test case is output. On the other hand, for path conditions 2), 3) and 4), the SMT solver generates 123, -2 and 3 respectively as a solution example, and treats this as the input data of test cases. In other words, three paths are extracted as in Figure 1, and executable test cases are generated as input data for variable *x*. As a result, all the paths that can be executed by the program in Figure 1 can be extracted as test cases. However, even using today's symbol execution, the extraction of test cases is not possible for all programs. This is due to three main issues.

1) Issue 1: Data type limitation

The data types that can be handled as symbols are limited to the numeric data type and string data type, and variable-length data structures such as arrays and lists are not supported. For this reason, it is necessary to introduce some mechanism other than symbolic execution in order to ensure test variations for these data structures, for example by preparing multiple fixed-length data structures before symbolic execution. The same is required for object type data.

2) Issue 2: Increase in analysis time

The required analysis time grows explosively as the scale of programs increases. Taking n as the number of conditional branches in a program, the number of paths that can be executed by the program is 2^n maximum, making analysis of programs of a practical scale virtual impossible. To solve this problem, algorithms designed to cover a large number of branch conditions with few paths, such as directed automated random testing (DART),[8] can be used to allow analysis in a practical amount of time. However, program structures that are difficult to cover exist for each algorithm, and therefore techniques for interrupting analysis along the way, such as limiting the search depth or implementing timeouts, are often used at the same time. In that case, conditional branches that cannot be covered in test cases remain.

3) Issue 3: Modeling of external world behavior

How to model the behavior of the external world on which the program to be tested is dependent is an issue. For example, most analysis engines that carry out symbolic execution can analyze only programs written in a particular development language. On the other hand, the behaviors of libraries, networks, OSs, database management system (DBMS), and so on, called by that program cannot be analyzed. Today's software seldom complete all the required processing with just one program, and thus it is essential that analysis be carried out taking into consideration the behavior of the external world upon which the program depends.

How to solve the above issues is a challenge that needs to be addressed to make practical application possible.

3. Efficiency improvement of unit testing

The most obvious avenue to achieve test-case generation using symbolic execution is to apply it to unit testing. Unit testing is a fine-grained test carried out during business application development. Code coverage, which indicates the ratio of the parts of the program that are executed by the test cases to the entire program, is often used as a measure of the completeness of a test. Test-case generation using symbolic execution is highly compatible with unit testing because it tends to increase code coverage. Of the unit test tasks, test analysis, test design (identification of test cases), test implementation (preparation of test data), and test execution, are automated, which allows develops to concentrate on the verification of execution results, resulting in greater efficiency.

The challenge that remains regarding the application of symbolic execution to unit testing is how to solve the three aforementioned issues. To deal with these issues, the authors adopted an approach constraining the program to be tested itself in a predefined manner and modeling the behavior of the external world as a dummy program (stub) for symbolic execution analysis. In today's business application development, the framework in charge of the system control portion and the programs that implement business logics are clearly separated. In terms of development man-hours, development of the business logic portion is most demanding by far, and the number of man-hours required for testing is correspondingly large. Taking note of this point, the authors decided to generate test cases by focusing on business logic programs. In this approach, the driver calling the business logic and the stubs of the framework and the application programming interface (API) of common parts called from the business logic are prepared without carrying out framework analysis. By carrying out analysis using symbolic execution combining a business logic program with the above, solving or mitigation of the above-noted issue is sought (**Figure 2**).

1) Solution for issue 1

As variable-length data structures cannot be treated as symbols, test variations must be prepared with a different mechanism than symbolic execution. To address this issue, the authors adopted the approach of preparing variations in terms of array and list length, and the use or non-use of objects, in advance in the drivers, stubs, and so on. For example, given methodA of the program to be tested, as shown in **Figure 3**, the generated testFunc driver method has two variations based on the number of elements of the list variable, which can be either 0 or 1. This is generated with a structure that switches the number of elements according to the value of boolean variable *c*. Actually, the parts recognized by the symbolic execution engine as branches that must be covered are the branches for variable *c*. By generating two cases that cover True and False for this branch condition, two cases are generated as a result, namely the case when the number of elements of the list object is 0, and that when it is 1.

As only predetermined variations as drivers or

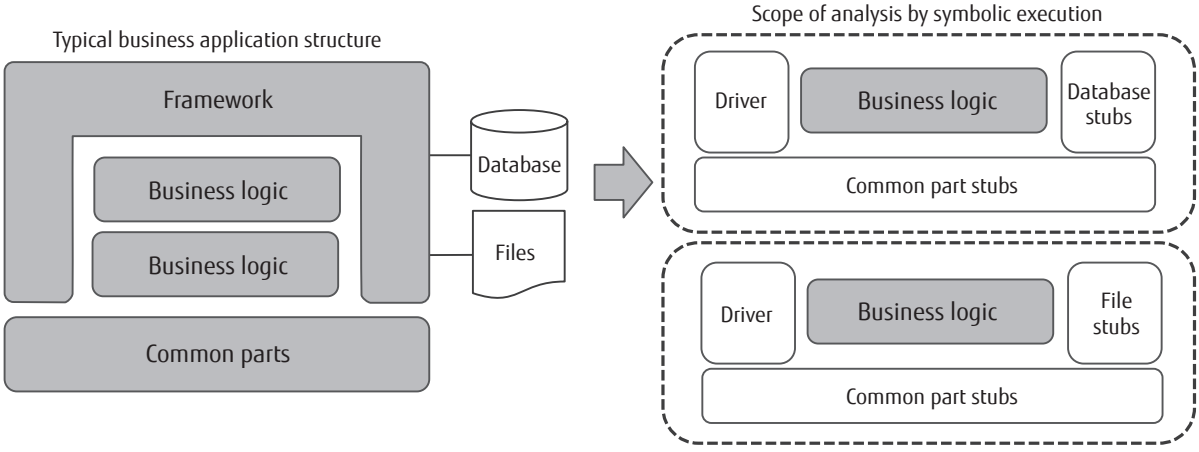

Figure 2
Driver/stub preparation according to framework.

Program to be tested

```
static void methodA(OrderBean order){
  List items = order.getItems();

  for (Item item: items)
...
}
```

Driver stub generation

Driver

```
Public class SymbolicDriver{
  static boolean c;
  public void testFunc(){
    OrderBean p = new OrderBean();
    ArrayList l = new ArrayList();
    if (c){
      l.add(new Item());//Number of elements = 1
    } else {
      // Number of elements = 0
    }
    p.setItems(l);
    Target.methodA(p);
}
```

Figure 3
Example of generated driver.

stubs are generated in this manner, a sufficient number of test cases cannot be generated for example for a program that performs special processing only when there are 5 elements. However, this approach works well for most business logic such as repeating some processing according to the number of elements. Those parts that could not be covered can be handled by adding test cases manually.

2) Solution for issue 2

The authors use search algorithms designed to raise code coverage, such as DART, so as to prevent explosive increase in analysis time. However, there are cases when test cases that cover branches cannot be discovered in loop structures such as "for." In order to eliminate such situations as much as possible, the logic for processing loop structures is broken down into components for reuse and implemented as APIs of the framework and common parts. On the other hand, during symbolic execution, analysis can be carried out by replacing these APIs with stubs that do not have a loop structure, which allows analysis of loop structures to be greatly minimized.

3) Solution for issue 3

The role of frameworks in business application development is to allow developers to concentrate on business logic implementation. For that reason, access to the external world, such as networks and databases, is provided by the framework in the form of APIs. Modeling of the external world consists solely in creating stubs for framework APIs, and is achieved by preparing in advance stubs that reproduce the behavior of framework APIs.

To verify the effectiveness of the approach proposed by the authors, researchers in Fujitsu Laboratories conducted an experiment using actual project assets.[9] The experiment was done using the following procedure, and the coverage of generated test cases and the man-hours required for each operation were evaluated.

1. Creation of framework stubs
2. Test-case generation and execution for task A programs
3. Test-case generation and execution for task B programs

1) Evaluation of coverage

Test cases with 100% code coverage were generated for 23 of the 26 functions of task A programs and task B programs. Regarding the remaining three functions, analysis of the parts that could not be covered revealed that they consisted of dead code that could not be executed regardless of the test case.

2) Evaluation of required man-hours

Table 2 lists the man-hours required for each operation. In the test, steps 1 and 2 were executed consecutively, and their man-hours are therefore combined. Moreover, the results were compared with the actual values when the same project assets were developed. Incidentally, in this development project, test cases were extracted manually.

Because man-hours were required for the creation of the stubs for the framework, the number of test man-hours for task A programs was 173% (2.73 times) of the actual value. However, the number of test man-hours for task B programs, which was performed reusing the already created framework stubs, was reduced by 39% compared with the actual value. Based on these results, once the framework stubs have been created by the developers of the framework, the number of man-hours expended by the developers of the business logic program can be expected to be reduced by 30% to 40%.

Table 2
Man-hour reduction through test-case generation.

	Number of code lines to be tested	Test time	Test time per 1000 lines	Ratio to actual test man-hours
Actual value for project	569 lines	13.0 h	22.8 h	–
1. Creation of framework stubs 2. Test generation and execution for task A	189 lines	11.8 h	62.4 h	173% increase (62.4/22.8=273%)
3. Test generation and execution for task B	57 lines	0.8 h	14.0 h	39% decrease (14.0/22.8=61%)

4. Application to regression testing

As mentioned in the preceding section, it used to be difficult to cover all conditions in a limited time, but owing to the efficiency gain obtained with symbolic execution, it has become possible to exhaustively generate test cases. As a result, the quality of unit tests can be improved in approximately the same time as before.

Another promising aspect of test-case generation using symbolic execution is quality improvement for regression testing. Regression tests are run when creating revised versions of programs, in order to check whether the modified program performs in the same way as the previous program. Specifically, regression testing consists in verifying whether, for a given test input, the output produced by the modified program is the same as that produced by the program before modification. The exhaustive extraction of test cases is very important for regression tests as well. This applies most particularly to when programs are modified, because the developer of the original program is often unavailable, making the identification of test case more difficult.

In addition to exhaustively generating test cases for pre-modification programs using symbolic execution, the automated method proposed by the authors records the output produced when test cases are executed as the expected values of test cases. This allows regression testing without manual labor by executing these test cases with the modified program. In unit testing, verification of the test execution results could not be automated and remained as a manual task for the developer. However, with regressive testing, result verification too can be automated, which is expected to yield dramatic returns in terms of improved efficiency.

To confirm the validity of this approach, an experiment of the above-described method was performed for the reconstruction of given product functions (C language, approximately 19 000 steps).[10] In this experiment, following test execution by the developer, test-case generation and execution was performed using symbolic execution. Following the detection and fixing of 27 bugs through the test performed by the developer, testing by the proposed method was able to detect an additional five bugs. This result indicates that rare bugs that cannot be detected through manual testing by a human operator can now be automatically detected through exhaustive search using symbolic execution.

5. Conclusion

This paper describes the various issues standing in the way of the practical application of exhaustive test-case generation using symbolic execution worked on by Fujitsu Laboratories, their resolution, and their application effect. At present, in addition to research to further improvement in the Java, C/C++, and COBOL[11] environments, Fujitsu Laboratories is conducting research and development work on test-case generation for applications that combine JavaScript and HTML in response to the great shift toward mobile in recent years.[12] Furthermore, beyond unit testing, Fujitsu Laboratories is planning to expand its activities to the area of integration testing for verifying system functions that are offered in combination with modules.

References

1) Association of Software Test Engineering (ASTER) Test tool Working group: Test Tool Beginning Guide (Introduction) (in Japanese). *http://aster.or.jp/business/testtool_wg/pdf/Testtool_beginningGuide_Version1.0.0.pdf*
2) K. Akiyama: Special Features: Hot Topics on Software Testing, 3. Combinatorial Designs for Testing Software.

IPSJ Magazine, Vol. 49, No. 2, pp. 140–146 (2008) (in Japanese).
3) T. Tamai et al.: Symbolic Execution Systems. IPSJ Magazine, Vol. 23, No. 1, pp. 18–28 (1982) (in Japanese).
4) A. Umemura: SAT/SMT solvers and their applications. Computer Software, Vol. 27, No. 3, pp. 24–35 (2010) (in Japanese).
https://www.jstage.jst.go.jp/article/jssst/27/3/27_3_3_24/_pdf
5) A. Orso et al.: Software Testing: A Research Travelogue (2000–2014). 36th International Conference on Software Engineering (2014).
6) Fujitsu Laboratories Ltd. et al.: Fujitsu Develops Software Verification Technology for Practical-use Web Applications.
http://www.fujitsu.com/global/about/resources/news/press-releases/2008/0404-02.html
7) Fujitsu: Release of business program development tool "Interdevelop Designer" (in Japanese).
http://pr.fujitsu.com/jp/news/2014/08/28-1.html
8) P. Godefroid et al.: DART: Directed Automated Random Testing. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05), pp. 213–223.
9) A. Katayama et al.: Symbolic execution verification trial on business system. IPSJ/SIGSE Software Engineering Symposium (SES2013) (in Japanese).
10) S. Tokumoto et al.: Enhancing Symbolic Execution to Test the Compatibility of Re-engineered Industrial Software. The 19th Asia-Pacific Software Engineering Conference (APSEC 2012).
11) Y. Maeda et al.: Test Case Generation by COBOL Symbolic Execution. 19th Foundation of Software Engineering Workshop (FOSE 2012) (in Japanese).
12) H. Tanida et al.: Automatic Unit Test Generation and Execution for JavaScript Program through Symbolic Execution. The 9th International Conference on Software Engineering Advances (ICSEA2014).

Tadahiro Uehara
Fujitsu Laboratories Ltd.
Mr. Uehara is currently engaged in research on software testing and Web API development technologies.