A Function Interaction Testing by Reusing Characterized Test Cases

Youngsul SHIN†a, Student Member and Woo Jin LEE†∗, Nonmember

SUMMARY This letter proposes a reuse method of unit test cases, which characterize internal behaviors of a called function, for enhancing capability of automatic generation of test cases. Existing test case generation tools have limits in finding solutions to the deep call structure of the source code. In our approach, the complex call structure is simplified by reusing unit test cases of called functions. As unit test cases represent the characteristics of the called function, the internal behaviors of called functions are replaced by the test cases. This approach can be applicable to existing test tools for simplifying the process of generation and enhancing their capabilities.

key words: test case reuse, test case generation, automation enhancement, characterization

1. Introduction

Although a program is performed by a series of functions and many errors are occurred during interacting functions, many researches focus on testing a single function itself, not on function interactions. And it is difficult to write significant test cases that verify interaction among the integrated source codes [1].

The code-driven automated tools for test case generation play significant roles in testing for large systems. They generate many test cases with high coverage automatically and fast. The code-driven tools for test case generation combine techniques from static program analysis, dynamic analysis, model checking, and constraint solving [2]. The test case generation tools explore all lines of the source code and compute inputs to drive the source code along specific execution paths.

Unfortunately, the constraint solver depends on its symbolic reasoning capability and suffers from the path explosion problem in the case of analyzing plenty of the integrated source code with the deep call structure [3], [4]. The model checker similarly suffers from combinatorial blowup of the state space, commonly known as the state explosion problem [5].

To relieve the path and state explosion problem, we propose to abstract function characteristics and reuse unit test cases. The abstraction of function characteristics generates test cases which reflect the internal behavior of the function conforming to each of test coverage criteria. At testing the integrated source code, the generated test cases used for testing a unit can represent and simplify the complicated source code of the unit. The tools for test case generation can easily treat the integrated source code that is expressed simply. Therefore, reusing the test cases supports for the test case generation tool to create fine test cases for integration testing according to each of test coverage criteria.

2. Characterization and Reuse of Test Cases

2.1 Characterizing Function Behavior by Test Cases

Characterizing function behavior generates test cases for a software unit according to each of test coverage criteria. A procedure to characterize the internal behavior of the unit consists of three steps: transformation to control flow graph, identification of test paths satisfying the selected test coverage criteria, and generation of inputs to execute the test paths. The source code is transformed to control flow graph which is represented as a tree. Once a test coverage criterion is selected, paths from the entry node to the leaf nodes are identified in order to satisfy the selected test coverage criterion. Before generation of inputs to execute the paths, the unnecessary nodes are eliminated using data dependency. Finally, given the paths, test cases to execute the paths can be generated using symbolic execution, linear programming.

The generated test cases abstract and represent the function characteristics according to each of test coverage criteria. A test case can satisfy multiple test criteria because the test coverage criteria have a coverage hierarchy as shown in Fig. 1. At testing of the integrated source code, a tester can select and reuse test cases that satisfy test coverage criteria for the development project.

Fig. 1 Abstraction hierarchy of test cases.
2.2 Reuse Mechanism of Characterized Test Cases

Reusing test cases represents and simplifies the integrated source code. The simplified source code does not have the complicated architecture that the test case generation tools cannot analyze. Therefore, fine test cases for testing the integrated source code can be generated by the tools since the complexity of the source code is decreased by reusing the test cases.

Assume that a tester verifies the function $A$ which calls the function $B$ which calls the function $C$ as shown in Fig. 2. The function $B$ should pass testing and is integrated with the function $A$ before testing the function $A$. Therefore, test cases that characterize the function $B$ integrated with the function $C$ are given at testing the function $A$. Reusing the test cases for the function $B$ can generate the representative code to abstractly represent the behavior of the function $B$. The representative code replaces the original function $B$ and is called by the function $A$ at testing the function $A$ as shown in Fig. 2. Excepting the function $B$ and $C$, the test case generation tools analyze only the function $A$ and the representative function of $B$ that simplifies the complicated interaction with the function $B$ and $C$. Test cases generated for the function $A$ are also reused to represent the function $A$ by other functions that call the function $A$. This process is repeated in incremental integration.

Assume that each of test cases that verifies different execution paths of the function $B$ is given as (1, 2) and (3, 7), where the first field of each tuple is an input and the second is an output. The behavior of the function $B$ integrated with the function $C$ is represented by the two test cases. Therefore, the simplified source code to represent the function $B$ is generated as shown in Fig. 3. As the representative function of $B$ delivers constant input values to the function $B$, the test case generation tools do not symbolically explore the function $B$ and $C$ and can easily analyze the representative function of $B$ that is simple. Without any change in the tools, reusing characterized test cases enhances the tools’ capabilities within the various domain of software except real time software since representative functions make additional call stacks.

3. Experiment

We compare the test coverage of test cases generated by the existing tools for automatic test case generation before and after applying our approach. Three tools, Randoop [6], CodePro [7], and CoView [7] are chosen and the statement coverage of their test cases is measured. The source code under test is Morfeo Openhealth project implementing ISO/IEEE Standard 11073-20601 [9].

In Fig. 4, we can see that applying our approach enhances 16% to 25% of the capabilities of the existing tools without any change in the tools, since reusing characterized test cases simplifies the process of complicated source code analysis of the existing tools.

4. Conclusion

This letter proposes a reuse method of unit test cases which characterize internal behavior of a called function. The experiment shows that the proposed reuse method enhances capability of automatic generation of test cases.

Acknowledgements

This research was supported by the Convergence Information Technology Research Center support program (NIPA-2012-H0401-12-1006).

References

[1] L. Mariani, F. Pastore, and M. Pezze, “Dynamic analysis for diagnosing integration faults,” IEEE Trans. Softw. Eng., vol.37, no.4, pp.486–503, July 2011.
[2] P. Godefroid, P. de Halleux, A.V. Nori, S.K. Rajamani, W. Schulte, N. Tillmann, and M.Y. Levin, “Automating software testing using program analysis,” IEEE Softw., vol.25, no.5, pp.30–37, Sept. 2008.
[3] F.A. Aloul, A. Ramani, K.A. Sakallah, and I.L. Markov, “Solution and optimization of systems of pseudo-Boolean constraints,” IEEE Trans. Comput., vol.56, no.10, pp.1415–1424, Oct. 2007.
[4] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst, “Finding bugs in Web applications using dynamic test generation and explicit-state model checking,” IEEE Trans. Softw. Eng., vol.36, no.4, pp.474–494, July 2010.
[5] F. He, X. Song, W.N. Hung, M. Gu, and J. Sun, “Integrating evolutionary computation with abstraction refinement for model checking,” IEEE Trans. Comput., vol.59, no.1, pp.116–126, Jan. 2010.
[6] Randoop, http://code.google.com/p/randoop, Nov. 2010.
[7] CodePro, http://code.google.com/javadevtools/download-codepro.html, 2010.
[8] CoView, http://www.codign.com/products.html, 2008.
[9] Source Code of Morfeo Openhealth Project, https://svn.forge.morfeo-project.org/openhealth, Nov. 2011.