Bug Localization by Constructing Reduced Traces

S. G. Groshev
Institute for System Programming, Russian Academy of Sciences,
ul. Solzhenitsyna 25, Moscow, 109004 Russia
e-mail: sgroshev@ispras.ru
Received December 28, 2007

Abstract—A method, based on the existing UniTESK test, which finds a bug in a system under test, is proposed to construct the minimal test that finds the same bug. This test can be used to localize this bug in the source code of the system. Two strategies for constructing such a test are considered, a comparative analysis of their advantages and disadvantages is performed, and the optimal strategy is proposed. A mathematical justification of the proposed method is given. An algorithm implementing this method is described, and its correctness is proved. An implementation of the proposed method for the CTESK testing tool is described.

DOI: 10.1134/S0361768809030037

1. INTRODUCTION
To satisfy the growing needs of users and maintain the steady development of modern society, complex software is required. The development of software involves multiple cycles of the form testing—bug detection—bug elimination—control testing. The more complex the software under development and the more critical tasks it is assumed to perform, the more different internal and external interactions can occur in it; as a result, more and more intricate bugs can appear. Some of them manifest themselves only in the case of complex combinations of the internal and external conditions; as a result, they are very difficult to detect, localize in the implementation, and correct; moreover, their potential harmfulness is greater.

There are testing technologies that enable one to test software with a prescribed completeness of the testing coverage. An example is the UniTESK technology for the automation of functional testing based on formal models that was developed in the Institute for System Programming of the Russian Academy of Sciences [1–4]. However, the more complex the system under test and the corresponding test suites, the more information about the observed behavior must be analyzed in the case when a bug is detected and the more difficult it is to understand which part of this information directly concerns the detected bug and which part does not. This paper is devoted to a method that makes it possible to localize bugs due to the automatic determination of the minimal conditions that are necessary to replay the bug.

2. STATEMENT OF THE PROBLEM
The methods used for testing depend on the complexity of the system under test. In simple software systems, there is no internal state or it is irrelevant for the testing task, and the response produced by such a system is unambiguously determined by the inputs. In this case, it is sufficient to repeat the same input to reproduce a detected bug; to localize the bug in the code, it is sufficient to check the code that is executed when the corresponding input is handled.

Much more often, the system’s behavior depends not only on the input itself but also on the system’s internal state, which, in turn, depends on the history of the interactions with the system. In this case, in order to reproduce a bug, one must somehow bring the system into the same state at which the system was when the bug was detected and apply the same test input. For the bug localization, one must check the code that handles the input in question in a particular state.

To test such complex systems, the approach based on modeling the system under test by a finite state machine (FSM) is widely used (see [5]). The states of such a machine correspond to the system’s states (usually, one model state corresponds to a class of implementation states), the input alphabet of the FSM corresponds to the set of all test inputs, the output alphabet corresponds to the set of all responses of the system, and the initial state corresponds to the state in which the system is at the beginning of the test execution. Under this approach, the problem of constructing test sequences is reduced to constructing a path in the transition graph of the FSM that satisfies the prescribed coverage requirements.

Arbitrarily, the testing methods based on FSMs can be divided into two classes—off-line and on-line testing [6, 7]. In the off-line methods, the state and the transition graph of the FSM are completely known in advance. On its basis, a test suite is first generated that provides the desired test coverage, and then the tests are executed. In the on-line methods, the state graph of the FSM is constructed step-by-step in the course of its traversal. The choice of the next test input is made based
on the information about the graph collected when performing the preceding test inputs. Such methods are more flexible, and they often considerably reduce the complexity of the test generation because the model FSM does not need to be described completely.

In the framework of the UniTESK testing technology, FSM modeling and on-line test generation are used. To create a test scenario, one only must describe a procedure for calculating the state of the model FSM given the implementation state and an enumeration procedure for the test inputs depending on the state [6]. All the work on designing the traversal of the FSM transition graph, which ensures the creation of specific test sequences, is performed by the library component of the testing system called the test engine. It automatically builds the transition graph of the model FSM step-by-step as it is traversed [5, 6]. The test scenario includes a function for calculating the state of the model FSM and a set of scenario methods. A scenario method includes a set of iteration variables (which may be empty), rules for enumerating their values, and a procedure for mapping the set of values of the iteration variables into a particular test input. The input alphabet of the FSM consists of ordered pairs of the form (name of the scenario method, set of values of its iteration variables).

In the course of executing a test scenario, UniTESK automatically generates the trace containing information about the operation of the testing system and of the system under test. Below, we assume that the trace contains information about a single test scenario and that the test scenario methods are executed one after another in a single thread. If the trace contains information about several noninteracting scenarios, we can single out the information about one of them and consider only this part. However, the proposed method cannot be applied in the case when the test inputs affecting the same state elements of the system under test are applied simultaneously in several threads [8].

To verify the correctness of the operation of the system under test, formal specifications of requirements are used. After each application of the test input, the response of the system is analyzed for the compliance with the requirements specification, and the new state of the model FSM is determined. The nonconformity of the observed response or state of the target system with the requirements specifications is considered to be an error, and it requires further analysis. As a result of the analysis, an error in the implementation can be identified, or it can be concluded that the specification model is incorrect. Below, we assume that the specification has already been debugged and is correct; therefore, all the discrepancies between the observed and specified behavior of the system under test are caused by implementation errors.

Often, the state of the system is very complex, and the only way of changing a state is to repeat the same test inputs that were used by the test. Therefore, a natural way to bring the system to the same state in order to reproduce the error is to completely or partially repeat the inputs that were applied by the test that detected this error. The sequence of inputs is taken from the trace generated by the test, and the process of their repetition is called trace replay. Due to the method used in UniTESK to describe the FSM, the trace replay is more reliable than the repeated test execution because the contract of the UniTESK test engine does not guarantee the same order of traversing the graph arcs.

Moreover, in real-life projects, there is often impossible to exactly determine the internal state of the system in the course of testing. In this case, it is modeled as a black box [6] whose state is not directly obtained from the state of the system under test but is rather calculated based on the history of interacting with this system. After each test input, the new state of the system is calculated based on the preceding state, the test input, and the system’s response. For various reasons (these can be both implementation and specification errors), the model’s state can cease to correspond to the implementation state in the course of time. This results in the discrepancy between the expected and the observed behavior of the system under test, and this discrepancy can manifest itself not immediately but only in the response to the subsequent test inputs; therefore, the moment of the occurrence of the error and the moment of its detection can be different. In this case, the replay and the localization of the error are much more difficult. Indeed, since we have no reliable knowledge of the system’s internal state at the time when the error occurred, the entire sequence of the test inputs registered in the trace of the test that discovered the error must be replayed, and the entire code that was activated by those inputs, as well as the elements of the implementation state that were affected by them, must be checked.

Aside from the advantages of on-line testing methods discussed above, they have some drawbacks. Often, off-line methods make it possible to produce a set of short tests that cover the entire functionality, and when a bug is detected using such a test, it is usually easy to localize. In distinction from the off-line methods, the on-line methods usually produce long tests. For example, the UniTESK test engine tends to construct a single path through the graph of the model FSM that covers all its transitions. As a result, the following situation can occur. By the moment a bug is detected, the UniTESK test has a chance to go a long and intricate way through the graph of the FSM performing many transitions and visiting some of the states several times. The analysis of the trace of such a test for finding a bug can be too complex; hence, we face the problem of finding the minimal test that can detect the same bug. To solve this problem, a method for constructing and replaying reduced traces was developed, which is described in this paper. Note