International Conference on Information Technologies (InfoTech-2010)

16th – 17th September 2010
Varna – St. St. Constantine and Elena resort, Bulgaria

The forum is organized in the frame of
"Dais of the Science of the Technical University-Sofia, 2010"
and unites the joint events:

24th International Conference on
Systems for Automation of Engineering and Research (SAER-2010)

6th International Workshop on
Technological Aspects of e-Governance and Data Protection (eG&DP-2010)

2nd International Seminar with Discussion on
Security Policy (Security-2010)

PROCEEDINGS

Edited by Prof. Dr. Radi Romansky

Sofia, 2010

Proceedings of the International Conference on
Information Technologies (InfoTech-2010)
16-17 September 2010, Bulgaria

AUTOMATED TEST CHAINING BASED ON COMPOUND STATES

Aleksandr Sukhorukov

Riga Technical University
e-mails: Aleksandrs.Suhorukovs@rtu.lv
Latvia

Abstract: The paper presents a method of construction of test suites as chains of tests based on their preconditions and postconditions modelled by name-value pair sets. Rules of test chain production are described. The concept of reset-test is introduced to make the method applicable in practice.
Key words: Automated testing, test suites, test sequence generation.

1. INTRODUCTION

Automated software testing has become a common practice in software development. Automated test script is a piece of software that performs some actions on a system under test and checks whether it behaves correctly. Test scripts usually can be executed automatically without human intervention. The complexity of automated testing resides in test script development and maintenance. If number of tests is large, it is hard to properly architect, design and structure the set of scripts (Berner *et al*., 2005).

Each well-defined test case has its preconditions (initial state) and postconditions (result state). We will define test suite as a set of tests, ordered in such way that postconditions of each test match preconditions of next test. Correctness of preconditions and postconditions are especially critical for automated tests, because if current state of a system does not match test preconditions, script will fail and it will lead to failure of all remaining scripts in a suite.

When tests are designed, the next step is to create a test suite, i.e. to order the tests so they can be executed sequentially. The intuitive example could be tests of creating, editing and deleting some kind of data in a system. This order seems to be

natural, however in more complex cases proper ordering of test in a suite can be a problem.

The goal of this work is to provide a method of test suite construction from sets of tests with preconditions and postconditions defined by means of name-value pair sets.

2. CONSTRUCTING TEST SUITES WITH SIMPLE STATES

Construction of test suite from available set of test scripts has similarities with automated planning of script event sequences (Memon et al., 2001) but instead of finding path to goal state it requires to find path including all tests of the available set.

Also for the addressed problem it is not important how test scripts were developed. They could be developed by hand or by applying various existing test case generation methods (Ferguson and Korel, 1996), (Godefroid, 2007). The same reasoning would apply to manual tests as well, but the problem is not so critical for manual testing, as human tester can observe discrepancy between test preconditions and actual state of system and resolve it before continuing.

The problem is more relevant to automated testing on graphical user interface (GUI) level. Tests on GUI level (Li and Wu, 2004) tend to be more complex than application programming interface (API) or unit level tests. Unit testing are more stable, easier to develop and maintain and there are many unit testing frameworks available helping to implement scripts for these tests (Hamill, 2004).

Our previous work on this problem focused on the case when preconditions and postconditions can be described as a simple state (Sukhorukov and Zaitseva, 2008). In this case each test script should contain metadata describing its initial state (preconditions) and final state (postconditions). If test script T1 starts at state *a* and finishes at state *b*, but test script T6 starts at state *b* and finishes at state *c*, it is possible to chain tests T1 and T6 into a suite. These relationships between scripts, states and suites can be modeled as an oriented graph, in which vertices represent system states, edges represent test scripts and each path in a graph represents a test suite. An illustration example of such graph is shown in Figure 1.

Construction of test suite on the graph means finding a cycle containing all edges. Some observations on test suite construction:

1. A cycle is required for test suite repeatability. If preconditions of the first test in suite would not match postconditions of the last tests, it would not be possible to repeat test suite execution twice without manual state change.
2. Edges are allowed to appear in suite more than once. As in Figure 1 example tests T16 and T15 have to be immediately followed by T17. It means T17 would appear at least twice in a test suite, and it is acceptable.
3. Though shorter suites (containing fewer edges) are preferable, it is not necessary to select the shortest possible one, as automated test execution

does not require human time. However the suite should be reasonably short, as a suite running for 3 days is unacceptable if 20 minutes equivalent is possible.

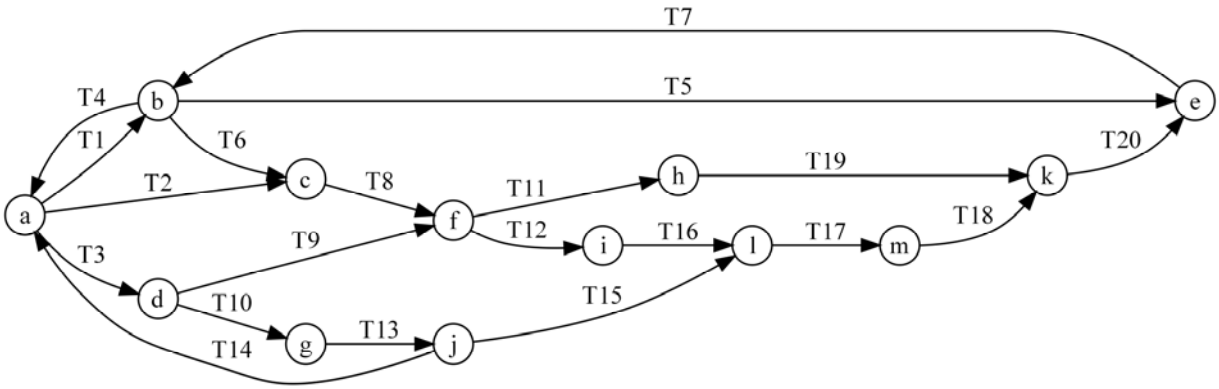

Fig. 1. Example of graph model of test set on simple states

It is easy to prove that necessary and sufficient condition for such cycle construction is that for each test there is a path from the initial system state to precondition state of the test and a path from postcondition state of the test to the initial system state.

Construction of full suite can be based on finding those paths and combining them to satisfy condition of every edge inclusion.

3. COMPOUND STATES OF NAME-VALUE PAIRS FORM

Test suites construction method based on simple states has limitations, especially in case of GUI level automated tests, therefore in this article we present a new method of test suites construction based on compound states.

A test aimed to verify whether system administrator is able to reset user's password could have several preconditions, for example:

1) user whose password has to be reset should already be registered in the system;
2) working user should be authenticated as system administrator;
3) user profile dialog form should be opened.

Such preconditions are hard to model with simple states. It is more appropriate to define state as a set of name-value pairs instead. Mentioned example precondition could be described as {'user A registered' = true; 'authenticated role' = administrator; 'active window' = user A profile}. We will call such name-value pairs state components. Simple states can be regarded as a trivial case of compound states with just one state component, for example {'current state' = a}.

It can easily be observed that state components relevant to some test may be irrelevant for another one. For example state component 'user A registered' is

irrelevant for test aiming to verify whether system administrator is able to configure e-mail notifications. There are three possible awareness types of test relation to state component. We will denote them as:

1. $R(x, y)$ – test requires a specific value x for this state component as a precondition and sets its value to y as a postcondition. It also could be that $x = y$ if test doesn't change the value.
2. $S(y)$ – test does not require a specific value for this state component as a precondition, but sets its value to y as a postcondition.
3. U – test is unaware of this state component. Any value will fit as a precondition and it remains unchanged as a postcondition.

We assume all tests to be deterministic, so if specific value is required as a precondition it either does not change or changes to another specific value, so it never becomes undefined.

Two tests T1 and T2 can be chained into suite if there is no conflict between any state components of T1 postconditions and T2 preconditions. If two or more tests can be chained, such chain will have its own preconditions and postconditions and hence awareness type. Awareness type of test chain can be found from awareness types of chained tests using Table 1.

Table 1. Chaining rules of awareness types

First test awareness type	***Second test awareness type***		
	$R(x_2, y_2)$	***$S(y_2)$***	***U***
$R(x_1, y_1)$	$R(x_1, y_2)$ *if* $y_1=x_2$ *otherwise – impossible*	$R(x_1, y_2)$	$R(x_1, y_1)$
$S(y_1)$	$R(x_2, y_2)$ *if* $y_1=x_2$ *otherwise – impossible*	$S(y_2)$	$S(y_1)$
U	$R(x_2, y_2)$	$S(y_2)$	U

For example, if test T1 does not require specific value of state component, but sets it, test T2 also does not have a precondition but sets the value, then it is the case $S(y_1)$ $S(y_2)= S(y_2)$ which means that chain [T1, T2] has no precondition, but its postcondition is value set by T2.

Chaining is not transitive however. If it is possible to create chains [T1, T2] and [T2, T3], it does not necessary mean that chain [T1, T2, T3] is also possible. For example, if awareness types of these tests would be respectively $S(y_1)$, U, $R(x_3, y_3)$, then all three tests cannot be chained if $y_1 \neq x_3$.

Chaining intransitivity means that graph-oriented approach will not work, as selection of test chain is more complicated than path selection in a graph. We propose a method of constructing test suite based on tree of chains:

1. A fictive zero-test will be used as a root of the tree. Zero-test models initial state of the system, defining initial values for all state components. Zero-test will have $R(x, x)$ awareness type for all state components if it is

required that test suite finishes at the same state it starts. So at the root of tree is chain consisting of one zero-test.

2. Each new level of tree is constructed from previous one. For each node of the level (representing a chain) it is necessary to find tests that can be appended to the chain. These new chains (having one more test at the end) become nodes of next level. This process is illustrated in Figure 2. The state consists of one state component *y* at this example for simplicity.

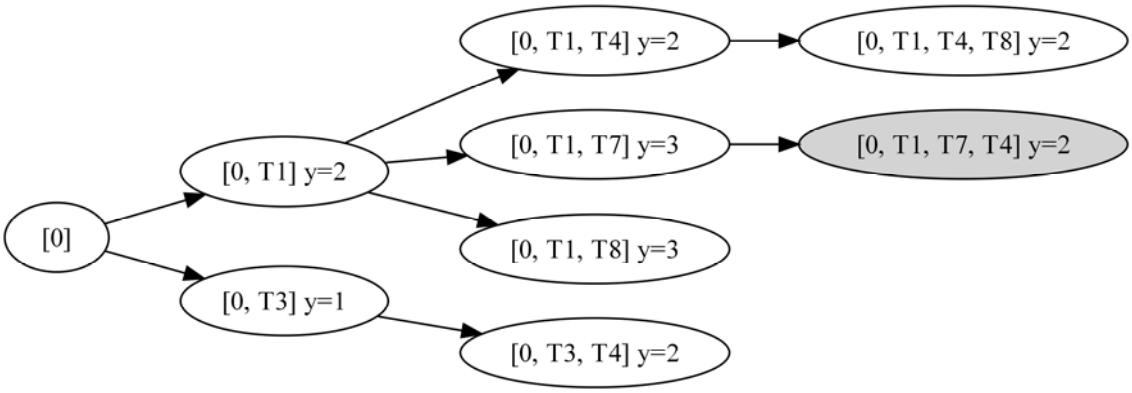

Fig. 2. Chain tree fragment

This process can be repeated until we find a chain containing all tests and finishing with zero-test. The algorithm has an exponential complexity and in practice when amount of tests is large this method in its pure form is inappropriate. We propose two heuristics that make the method more practical.

The first heuristic is introduction of reset-test. Usually in practice there is a universal method to reset system state to the initial one. If it is the true, the zero-test can be implemented as a reset call. Awareness type of such reset-test would be $S(x)$ for every state component. It should not be candidate for appending to existing chains. It means that the tree should stop growing once every test was included at least at one chain at any level. Then it is relatively easy to select a minimal set of chains and concatenate them. Reset-test assures that those subchains start at initial system state.

The second heuristic is practical when the first is applied. For large number of tests the tree may quickly grow in breadth. If two or more chains appear for which the last test is the same and the postconditions of the whole chains are the same, then only one of such chains (the shorter one) should grow further. In Figure 2 example a chain is highlighted that will not grow further. Its last test is T4 and postconditions are {y=2} and there is one more tree node with the same last test and postconditions. Subtrees of these two nodes would be equivalent and would reach the same tests.

4. CONCLUSION

The proposed methods of test suite construction can be used in automated test frameworks as a part of execution schedulers.

Compound states of name-value pairs allow modeling more complex cases than simple states do. However, this approach has limitations as there are cases when fixed values of state components are inappropriate. For example, test could increase value of state component by one. The proposed method does not allow such postconditions and should be further enhanced.

The proposed algorithm has exponential complexity, although two heuristics proposed make the method practically usable. Our experiments show that construction of test suite for hundreds of tests usually takes a few seconds on average desktop computer.

This work has been supported by the European Social Fund within the project „Support for the implementation of doctoral studies at Riga Technical University".

REFERENCES

Berner, S., R. Weber, R. K. Keller (2005). Observations and Lessons Learned from Automated Testing. In: *Proceedings of the 27th international conference on Software Engineering*, pp. 571-579. ACM, NY, USA.

Ferguson, R., B. Korel (1996). The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology, 1(**5**), pp. 63-86.

Godefroid, P (2007). Compositional dynamic test generation. In: *Proceedings of the 34th Annual ACM Symposium on Principles of Programming Languages*, pp. 47-54. ACM, NY, USA.

Hamill, P (2004). *Unit Testing Frameworks*. O'Reilly, CA, USA.

Li K., M. Wu (2004). *Effective GUI Test Automation: Developing an Automated GUI Testing Tool*. Sybex, CA, USA.

Memon, A., M. Pollack, M. Soffa (2001). Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 2(**27**), pp. 144-155.

Sukhorukov, A., L. Zaitseva (2008). Automated Test State Management Framework. *Scientific Proceedings of Riga Technical University*, Series 5, Volume **34**, pp. 215-224.