# Model Driven Test Adaptation as Repair

Sergiy Boroday[1], Hesham Hallal[1]

[1] CRIM, 550 Sherbrooke West, Suite 100
Montreal, H3A-1B9, Canada
{boroday, hallal}@crim.ca

**Abstract.** Test cases are often modified to conform to changes in software. In this paper, test case adaptation is cast as a model-driven program repair problem. Such a view allows one to address even the most complicated test scripts (testing programs) with non-trivial control and data structures. At the same time, we instantiate our approach for sequential preset tests cases. Based on our view on adaptation as test repair, the problem of preset sequential test adaptation could be solved with well known techniques developed in the context of spellchecking and code correction. A preliminary case study on a travel request management business process is reported.

**Keywords:** Test Adaptation, Repair, Edit distance, Spellchecking

## 1 A Novel Approach to Test Case Adaptation

In a modern business environment, software is constantly modified to respond to technological and business changes. Each time software is updated, it must be tested and validated. However, some test cases could become invalid and need to be updated.

The majority of existing approaches to adapt test suites to program changes tend to derive new test cases to replace the obsolete ones rather than trying to actually adapt them. In [1], authors target adaptation of exhaustive test suites. However, exhaustive test suites are not yet sufficiently accepted by test engineers due to computation costs. More often, the test objectives are somehow re-engineered, and new test cases are produced according to these objectives. Typically, the new test cases just try to preserve some structural coverage metric. However, test designers do not always follow well-established coverage criteria and could be driven more by experience and intuition. Moreover, previous work on test suit reduction [2] shows the same coverage does not guarantee the same fault detection effectiveness of these suites.

One obvious approach to test case adaptation, rather than replacing obsolete test cases, is to keep old input part of the test case, but update outputs according to the new model [3]. Obviously, this is not the best adaptation since old inputs could be simply rejected or even halt the SUT, so no further inputs could be applied. Thus, the input part of the test might also need to be adapted.

Here, we propose a novel approach to adapt test case to changing software while preserving as much of the original test case as possible. Our approach is based on the emerging theory of model driven program repair, which aims at automatic recognition and bug correction in programs, in order to fulfill some objectives such as satisfaction of a requirement or conformance to a predefined model. Usually, a minimal repair, which introduces as little changes as possible is sought. The repair is defined as a sequence of basic corrections on the program (script in our case). Of course, brute force algorithms, which try all the possible repairs do not scale up well. Thus, program repair techniques are based on game theory [4], abductive theory revision [5], and on heuristics. Related ideas are also discussed in intelligent debugging [6], hardware/robotics, self-repair/self-healing context, though, in these areas, highly specialized technical methods prevail over general and externalized model-based repair methods.

We consider reuse of ideas from program repair theory to test adaptation promising since a contemporary test case is often a program (script) itself. While test scripts are rather simple programs, still, they could encompass parameters, complex data structure, components, and even some control elements such as branches, loops etc. In the test adaptation setting, the model of the (new version) of the program becomes the objective. Then, the test is adapted to conform to the new model of the program, using a program repair tool or technique.

The model of the modified system should be specified by test designers, e.g., by changing the old model according to the expected changes. Alternatively, one can try to infer such a model using static or dynamic methods. In an extreme case, the code of the new program could be considered as a test property "program should pass the test". In such a case, the updated test will be executable, however the correctness of the incorporated oracle (expected outputs) is never guaranteed.

In the next section we instantiate our approach for a simple and intuitive case, when a test case is just a sequence of inputs and outputs rather than a program. While a more general case may require sophisticated methods of game theory or AI, for the sequential case, we show that test repair could be reduced to a well known problem studied in the context of spellchecking.

## 2 Test Repair as Spellchecking (Minimizing Edit Distance)

Let a system under test (SUT) be (output) deterministic, modelled by a Petri net or an automaton with input and output actions (with no data), and a test case be simply a word in the language of the automaton or Petri net. Petri nets are mentioned due to their increased use in business process modeling since they simplify representation of constructs like parallelism and choice. Basic correction operations are *insert*, *delete*, and *replace* of a single action. In this setting, adaptation of an obsolete test case boils down to finding a word of the language of the modified SUT model with minimal Levenshtein edit distance [7] to the obsolete test case. Unlike general program repair, the problem is well studied due to numerous applications to spellchecking, code correction, and molecular biology. For the problem of computing Levenshtein distance and finding such a word in a regular language efficient algorithms are known [8], [9]. For context-sensitive languages, which strictly include Petri net languages, these problems are computationally hard; however, efficient parallel algorithms are known for context-free languages [10]. In a pragmatic setting, fast correction algorithms developed in the context of spellchecking may be even more appropriate [11].

Other edit distances, such as Hamming distance, Damerau-Levenshtein distance, Wagner-Fischer distance, or Jaro-Winkler distance could also be considered, however, Levenshtein distance corresponds directly to our problem statement, i.e., finding a minimal test repair that involves only three basic correction operations: insert, delete, and replace.

## 3 A Case Study

We consider a small example to illustrate the adaptation problem and the proposed solution. The example involves a model of a business process that takes care of making travel arrangements for an employee within a company. The process includes two sub-processes, booking a flight and a hotel. The executions of the two sub-processes can interleave. The processes are represented by a Petri Net, since to derive these processes we used the Petri Net based process mining tool ProM [12].

In the first model, called the original model *O*, the following activities are involved: *make a travel request* (*mtr*); *select flights* (*sf*), *select hotels* (*sh*); *choose one flight* (*cof*); *select one hotel* (*soh*); *book flight* (*bf*); and *book hotel* (*bh*). The Petri Net of *O* is shown in Figure 1. Note that the activities in the process are not necessarily performed following the above order. The figure shows that some activities are actually concurrent and can appear in any order. For example, the activities *select hotels* and *select flights* are concurrent while *select one hotel* always occurs before *book hotel*.

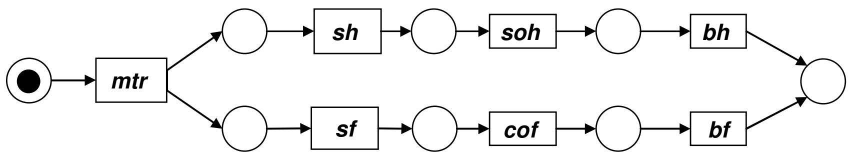

Fig. 1. The Petri Net of the original model.

The second model, inferred from the log files produced by the same business process after some changes is the modified model *M*. This model is the Petri Net in Fig. 2, where the following changes are identified:

1. The activities *issue a paper ticket* (*ipt*) and *issue an electronic ticket* (*iet*) are added to the sub-process booking a flight. One of the two activities is performed in an execution of the process. They represent a choice for the user of the system. However, the chosen activity must occur before booking the flight.

2. The activity *choose one hotel* of the sub-process booking a hotel is replaced by the activity *choose room*.

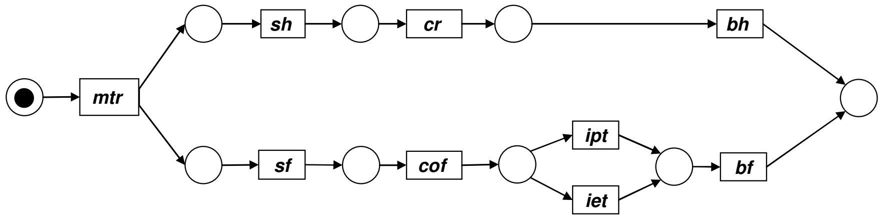

Fig. 2. The Petri Net of the modified model.

We discuss the effects of the changes introduced to the business process on testing the implementation of the process. Consider the original model whose language includes the word *mtr sh soh bh sf cof bf* that features all the activities of the process. This word, which can be seen as a test case of *O*, is no longer valid in the modified model because:

1. The language of *M* does not include words with the activity *select one hotel* (*soh*).
2. Either *issue a paper ticket* (*ipt*) or *issue an electronic ticket* (*iet*) must feature in any word of the language of *M* that includes all the activities of the process.

The test case can be adapted (repaired) by applying the following basic corrections:

1. Replace *soh* by the new activity *choose room* (*cr*).
2. Insert either of the two activities *ipt* or *iet* after the activity *cof*.

For this simple example, there are two possible adaptations of the test case, namely *mtr sh cr bh sf cof iet bf* and *mtr sh cr bh sf cof ipt bf*. Note that both adaptations are at an equal Levenshtein distance from the original test case (namely, two). Any other word would be at Levenshtein distance greater or equal to four and might heavily distort the original objective of the test designer.

## 4 Discussion

Here we discuss the limitations of the proposed approach and the possible workarounds. If the minimal repair of test case changes it drastically or simply deletes it, we believe that the test case should not be repaired. Instead, it should be replaced. Similarly to spellchecking some threshold of tolerance could be defined. When an SUT undergoes some cardinal changes, the approach might even be no longer applicable.

The proposed approach does not necessary guarantee the adapted test case to have the same structural coverage as the original test case. Combining test adaptation with structural coverage constitutes our future work.

While data structure repair [13] is occasionally considered in the literature, adaptation of the data part of test cases could be more challenging task.

## 4 Conclusion

The test adaptation problem is cast as a program/system adaptation problem. Difficulties and advantages of the proposed approach are discussed.

**Acknowledgments.** We thank A. Petrenko for stimulating discussions.

## References

1. El-Fakih, K., Yevtushenko, N., Bochmann, G.: FSM-Based Incremental Conformance Testing Methods. IEEE Transactions on Software Engineering 30 (2004) 425-436
2. Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C.: An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. Software Maintenance., Bethesda, Maryland, USA (1998) 34-43
3. Fraser, G., Aichernig, B., Wotawa, F.: Handling Model Changes: Regression Testing and Test Suite Update with Model-Checkers. Model Based Testing Workshop, Braga, Portugal (2007) 29-41
4. Jobstmann, B., Griesmayer, A., Bloem, R.: Program Repair as a Game. Computer Aided Verification, Edinburgh, Scotland, UK (2005) 226-238
5. Buccafurri, F., Eiter, T., Gottlob, G., Leone, N.: Enhancing Model Checking in Verification by AI Techniques. Artificial Intelligence 112 (1999) 57-104
6. Stumptner, M.: A Survey of Intelligent Debugging. AI Communications. 11 (1997) 35-51
7. Levenshtein, V.I.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady. 10 (1966) 707
8. Wagner, R.A.: Order-n Correction for Regular Languages. Communications of the ACM. 17 (1974) 265-268
9. Konstantinidis, S.: Computing the Levenshtein Distance of a Regular Language. Information Theory, Awaji Island, Japan (2005) 108-111
10. Pighizzini, G.: How Hard Is Computing the Edit Distance? Information and Computation 165 (2001) 1-13
11. Schulz, K.U., Mihov, S.: Fast String Correction with Levenshtein Automata. International Journal on Document Analysis and Recognition. 5 (2002) 67-85
12. ProM toolkit. Process Mining Group, IS subdepartment, Eindhoven Technical University (2007) http://is.tm.tue.nl/~cgunther/dev/prom/
13. Demsky, B., Rinard, M.: Automatic Data Structure Repair for Self-Healing Systems. First Workshop on Algorithms and Architectures for Self-Managing Systems, San Diego, California (2003) 78-95