Finding Common Subsequences in Recorded Test Cases

Martin Filipsky, Miroslav Bures and Ivan Jelinek
Department of Computer Science and Engineering
Czech Technical University in Prague
Prague, Czech Republic
{filipma2, buresm3, jelinek}@fel.cvut.cz

***Abstract*—Current trends in the agile software development prefer to deliver finished stories with automated tests, which results in a fact that many Quality assurance engineers struggle with the lack of time. Rapidly changing applications prevent them from finishing the automation by the end of the sprint as they cannot develop the tests in advance, and have to wait until the stable deliverable is done. Test recording might help them to resolve the problem as it offers very fast test automation in comparison to other approaches. However, it results in a very expensive and a time demanding test maintenance. In this paper, we present an approach that helps the engineers with the maintenance by introducing a concept of automatically detected reusable parts within the test recordings. Those reusable parts increase the efficiency of the test recording approach, remove its main drawbacks, and help to bring test recording closer to scripting approaches.**

Keywords-functional testing, test automation, test recording, genetic algorithm

I. INTRODUCTION

Test automation includes a couple of challenges [9]. Since testing teams are usually limited by finances, time as well as resources [3], they have to use simple but efficient approaches for the test harnessing. Here comes the test recording [5] in place as it allows creating automated tests quickly. On the other hand, this method is not generally understood as efficient due to its significant maintenance overhead [1].

In our recent research [6], we have proposed a framework for the test automation based on the test recording. We introduced a concept of reusable parts allowing simplifying the test maintenance. Introducing the reusable parts means to find common parts within the recorded tests. The problem of finding them can be transformed into the finding longest common subsequence problem [2].

The paper is organized as follows. Section 2 introduces the problem. Section 3 summarizes the previous results. In Section 4, we describe our solution of the problem. In Section 5, we conclude with outlines for future work.

II. THE PROBLEM

The Longest Common Subsequence (LCS) problem is defined as finding LCS common to all sequences in a set of sequences. The subsequence is a sequence that can be derived from another sequence by deleting some elements of the original sequence without changing the order of the remaining elements. Unlike the subsequences, the substrings cannot be derived from another string by deleting some elements.

Consider a string S = "AACECAACE", then following strings: (i) S, (ii) "AACAE", (iii) "CCCE", (iv) "ACECA" and (v) ε are subsequences of the string S. The subsequence "ACECA" is also the substring of the string S but the subsequences "AACAE" and "CCCE" are not. Now consider strings S_1 = "**A**EB**EEBC**C**BA**C**A**" and S_2 = "CE**ACEBEBCBAA**". Then "**AEEBCBAA**" is the LCS of the given strings, which currently preserves the order of elements and allows deleting elements from the original strings.

The standard LCS problem is defined for finding a single LCS. However, if we need to find all subsequences with at least length l, the problem is getting more complex. In general, the decision, if a subsequence w, which is common to all sequences and has the length at least l, exists over an alphabet Σ, is an NP-complete problem [13]. To overcome this limitation, we are planning to employ an evolutionary computational technique to find LCS.

Understanding tests as sequences of steps might be more beneficial than understanding them as strings. Finding common subsequences (CS) might result in longer subsequences than finding common substrings. However, it brings the need to define conditions when a subsequence is valid when excluding some steps from the test case. Otherwise, it might happen that the found CS could not be executed independently as the some steps might depend on excluded steps. Therefore, the state of the application would not be identical for all steps within the common part.

When finding CS for informational purposes, all steps can be excluded. However, if we want to understand CS as functions (as we want to get closer to the scripting approach), we have to exclude all steps changing the state of the application (Fig. 1), i.e., only the passive (validation) steps can be interposed between the common sequence.

III. RELATED WORK

Searching in structured data like test steps or test scripts represent challenges in the current computing. As the machine processing becomes more widely used in order to replace the human labor, standard approaches [10, 11] for the string searching introduced in 70's cannot be often easily employed for those data.

Unlike unstructured data, the structured data are organized in elements. However, the elements (tags) are not supposed to convey information, e.g., in Extensible Markup Language (XML).

 ISBN: 978-1-61208-304-9

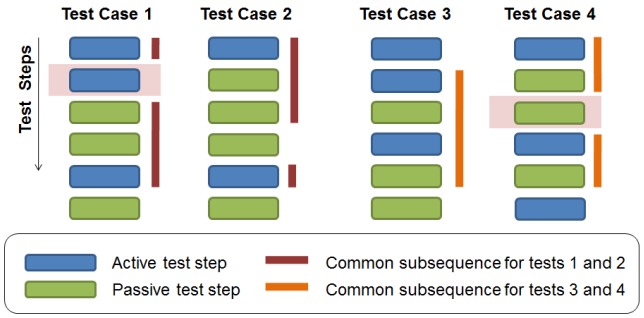

Figure 1. Two types of inserted steps (in light red)

Tags define a structure of the document. We talk about hybrid data when both types of data are in one document.

Zhu et al. [17] noticed that the text search on hybrid data may result in a bad ranking of the searching results. They demonstrated why the text search fails or gives insufficient results when used without considering the structured data.

XML can be seen as a good format for a test case representation. However, searching within those structured data requires special approaches, which can be divided into two categories: (i) information retrieval, and (ii) database-oriented. The database-oriented approach [12] is based on a decomposition of XML documents and their storage in relational databases. The drawback is a query processing, which may become expensive due to an excessive number of joins required to recover information from the fragmented data. The information retrieval approaches employ other computational techniques like genetic algorithms in several ways [16].

Srinivasa et al. [15] introduced an approach for an XML information retrieval mechanism. Based on keyword queries, they explored how to retrieve and rank XML fragments using Genetic Algorithms.

An evolutionary technique for the LCS problem is discussed in [7]. The genetic algorithm (GA) encodes candidate sequences as binary strings as long as the shortest of given string. Authors initialize conventionally random genotypes. They demonstrated that the algorithm always found an optimum solution, runs in reasonable times even on large instances, and achieves better results when compared to approaches based on the dynamic programming.

Julstrom and Hinkemeyer [8] noticed that GA might find good solutions more quickly in situations, when a problem is one of constrained optimization, and genotypes of the initial population are represented by empty solutions.

Finding longest common subsequences in strings is commonly solved by GAs or dynamic programming. The recent research shows that GAs achieve the best results in comparison to other approaches. Several research teams presented approaches finding the LCS in strings. Nevertheless, those approaches do not deal with structured and parameterized data represented by tests in different input alphabets. Since the current research in testing is mostly focused on the generation of test cases based on a code analysis [4], or on an analysis of regression test selection [14], we see a potential in the research of techniques for the maintenance of recorded tests to decrease costs for the test maintenance.

IV. Proposed Solution

In this section, we present individual parts of our approach. We start with mapping tests to strings. Then we present control parameters, outputs, and introduce our proposal of LCS solver. Finally, we explain step signatures.

A. Mapping of Tests to Strings

Current solutions for the LCS problem are proposed for strings (unstructured data). Since test cases are represented, e.g., in a domain-specific language (DSL), we need to adapt the current solutions to work with the structured data. Strings consist of single elements, i.e., characters, which form sequences. We plan to represent the test cases internally in the DSL (see Listing 1) describing tests, modules, objects, actions, etc. The Listing 1 shows the recorded user activity forming the sequence in the XML.

If we consider all child tags of the XML tag Step including their parameters and values, we will deal with high number of variables. It will result in a difficult mapping of the XML tag Step to a single character required by LCS solvers. On the other hand, if we consider just Step as one character, we can understand the tag as one character of the string. Therefore, the string will consist of complex units (Fig. 2). Such a representation enables working with structured data using conventional LCS solvers. However, this approach would be too simplified as the steps might be understood as identical. They do not have enough properties for the identification, since the tag id or tag name is not enough. Therefore, all steps mapped to the same character could not be recognized. To identify test steps, we introduce step signatures, which are supposed to replace a step description. Otherwise, we would have to choose between the full text search not recognizing two identical but parameterized steps, and the mapping of steps to characters not allowing distinguishing them.

Unlike test cases in DSL, the use of, for example, Java brings new challenges. First of all, steps represented by commands or functions of the scripting language have to be simplified. Consider the complexity of the comparison when counting with language-specific features, parameters etc. However, the simplified elements still should have signatures to describe them, which results in a need to find either direct mapping of commands to steps including signature definitions for every proposed language. Another option is to find a general mapping of a limited subset of commands to the intermediate layer (DSL), and propose one signature based on the DSL.

In our research, we plan to do more investigations in order to decide if it is better to work with the source code directly, or if it is worth to transform the source code to the DSL and then, to process this representation.

B. Inputs and Outputs

The LCS solver expects two kinds of input data: (i) raw input data intended for processing (test recordings), and (ii) control data driving the processing. For the finding LCS, we expect to provide the LCS solver, i.e., the GA, with the input files either in the DSL, or in direct source codes.

 ISBN: 978-1-61208-304-9

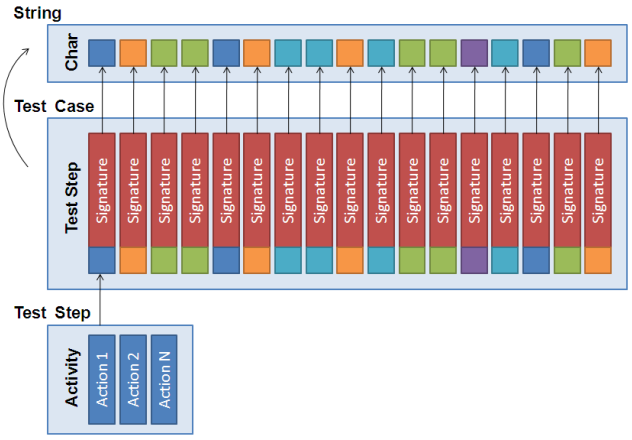

Figure 2. Mapping tests to simple strings

The condition is that the relevant mapping exists from test scripts to elements with signature. The output from the LCS solver should be a processed package of test recordings with identified common subsequences.

Since the LCS solver should be proposed to find the longest common subsequence as well as shorter CS in order to detect reusable parts, we need to provide the LCS solver with a threshold defining what lengths of common subsequences we are interested in. Moreover, we want the LCS solver to work with simple test step signatures and/or with the complex ones allowing recognizing identical but parameterized steps. In other words, the LCS is supposed to work at different level of details.

C. Evolutionary Computations

We based our solution on the approach presented in [8] and tailored the GA used in the LCS solver to fit our needs. For the LCS search, candidate sequences are encoded as binary strings as long as the shortest mapped given tests or the first given test if they are of the same length. If the element is present in the candidate sequence (in the chromosome), it is encoded by "1". If not, it is encoded by "0". Since [8] demonstrated that the GA achieves better results when the population is empty, i.e., the population is represented by zeros, we have decided not to employ any technique for the generation of the population.

Consider the example of three mapped tests T_1 = "**A** E B **E E B** C **C B**", T_2 = "C E **A E E** C **B C B**", and T_3 = "**A E E** E A **B C B** E", and the chromosome c[*] = 1 0 0 1 1 1 0 1 1, then it means that T_1: c[i] = 1 is in the subsequence T_1[i], and T_1: c[i] = 0 is not in the subsequence T_1[i]. T_1 represents the shortest given test or the first test from tests with identical lengths.

Once GA finds a solution of the LCS problem, the LCS will be encoded in the chromosome. However, the found solution represents the LCS in one test, but does not define where to find the subsequence in other tests. We only know the mapping from the chromosome to T_1. Since the LCS solver is required to build a structure enabling to identify and access the subsequence in all tests, the computation of the LCS has to be followed up by another stage of computations finding the mapping.

The fitness function v is proposed to remunerate (1) long sequences, (2) the genotype whose subsequence is long as T_1, (3) strongly remunerate the genotype, in which the subsequence appears for each given test (4) strongly penalize the genotype whose subsequence is not found in any test. The fitness function cannot be positive unless the sequence appears in all given tests. Based on the assumptions above and the research of [8], the initial general fitness function is defined for every case as follows:

$$v = l + \alpha * m \quad (1)$$

$$v = v + \beta \quad (2)$$

$$v = \gamma * v \quad (3)$$

$$v = \delta * v * (K - m) \quad (4)$$

where l is a length of the subsequence, which $c[*]$ represents, m is number of tests, which match with the subsequence, and K is the number of tests in the instance. The constants α, β, γ, and δ represent the parameters of the genetic algorithm and will be experimentally determined.

We are planning to employ several techniques for driving the evolution of the population, which will be divided into elite genotypes and the majority population. If the elite population does not change for several generations, some of the elite genotypes will be replaced by random genotypes to avoid local optimums. Remaining genotypes will be evolved using either a selective breeding of a position, or a mutation of the position. The genotypes to be modified will be selected by the tournament selection with the probability $1/l$. We are intending to carry out additional investigations to decide which strategy would bring the best results.

D. Signatures

Steps of parameterized tests can be compared only in text mode. Therefore, we proposed signatures to help the steps get compared, and find common parts. Since the structure of the command might be variable (for example, consider commands with one, two, or more parameters), the usage of regular expressions would require to define regular expressions for all possible combinations to compare strings. Otherwise, the standard LCS solver could not compare parameterized data. Unlike the regular expressions, the signatures allow to define simplified ones for rough searches, and also detailed signatures for fine-grained searches. Moreover, they make use of the opportunity of the clear structure of the DSL (see Figure 3 representing a sample recorded test), and can be built in a simplified way for all commands than regular expressions.

```
<test case id=1 name="AddBob">
  <step id="1">
    <object id="Menu" type="Tree"\>
    <action name="Select" onFailure="">Tools;Login</action>
  </step>
  <step id="2">
    <object id="Username" type="InputBox"
      environment="Flex" \>
    <action name="Set" onFailure="">Alice</action>
  </step>
  <step id="4">
    <object id="Login" type="Button" environment="Flex" \>
    <action name="Click" onFailure="" \>
  </step>
</test case>
```

Figure 3. Recorded test case in the DSL

 ISBN: 978-1-61208-304-9

Let us explain the signatures on the example of the recorded test case captured in the DSL. The test case represents the login to the system. The base signature consists of descriptions of two entities (objects and actions). We proposed several levels of signatures for different needs. The Level 0 signatures are intended to represent subsequences of similar objects and actions. It provides the users with a possibility to find groups of similar commands independently of concrete objects. Level 1 is proposed for the standard LCS search. It enables to work with parameterized tests, but it is not so strict like Level 2, which finds absolute conformities of the subsequences including input values. Level 2 gives the user a choice, what attributes and parameters should be in the signature.

TABLE I. SIGNATURES

Level	Step	Signature
0	2	obj:inputbox&act:set
1	2	obj:inputbox.username&act:set
2	2	obj:inputbox.username+environment=flex &act:set+val:(hash)

The Table 1 presents the signatures for each level based on the sample recording (Figure 3) for the step 2. To simplify the signature as much as possible (consider long input data), the input parameters are replaced by hashes. The syntax of the signature is defined as follows:

obj:<type>{.<object_name>}{+<attribute>=<value>}&
act:<action_name>{+<parameter>:<value>}

where *obj* stands for the object entity, *act* represents the action entity, the & char links different entities. If more attributes are required to describe entities in the signature, they can be associated with the entity using the char "+". The entity attributes are not mandatory.

V. CONCLUSIONS AND FUTURE WORK

We have proposed the approach for finding LCS of test steps based on the evolutionary computational approach presented in [8]. Moreover, we proposed the method of the adaptation of the GA processing strings to process structured data represented by test cases. Furthermore, we introduced signatures for descriptions of steps, which currently enable finding LCS in different equivalence classes.

Our next goal of the research is to conduct experimental verifications of the proposed approach as well as to tune up the parameters of the GA. We are planning to compare results gained using the signatures to results gained using the regular expressions, and to find out the impact of different sizes of the input alphabet. One of our goals is also to confirm or disprove whether it is better transform inputs into the DSL, or if it is worth to work with test recordings directly without preprocessing.

Finally, we are planning to evaluate the results from several points of view. Firstly, we will check whether the results make sense, and whether found LCS would be similar to reusable units designed by human testers. Secondly, we will investigate the contribution of such approach with an emphasis on the efficiency of test automation and test maintenance.

REFERENCES

[1] B. R. Anand, H. Krishnankutty, K. Ramakrishnan, and V.C. Venkatesh, Business Rules-Based Test Automation: A Novel Approach for Accelerated Testing, White paper, Infosys, SETLabs Briefing, Special Issue April 2007, pp. 21-28.

[2] M. F. Balcan, CS 3510 – Design and Analysis of Algorithms, Lecture notes, Georgia College of Tech Computing, 2011.

[3] R. Black, Investing in Software Testing: The Cost of Software Quality, White paper, RBCS, 2000.

[4] Ugo Buy, Alessandro Orso, and Mauro Pezze. 2000. Automated Testing of Classes. In Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '00), ACM, New York, USA, pp. 39-48.

[5] M. Fewster and D. Graham, Software Test Automation: Effective Use of Test Execution Tools, Addison-Wesley Professional, ACM Press Books, September, 1999.

[6] M. Filipsky, M. Bures, and I. Jelinek, Framework for Better Efficiency of Automated Testing, In Proceedings The Seventh International Conference on Software Engineering Advances, Lisbon, Portugal, 2012, pp. 615-618.

[7] B. Hinkemeyer and B. A. Julstrom, A Genetic Algorithm for the Longest Common Subsequence Problem, In Proceedings of the 8th annual conference on Genetic and evolutionary computation, GECCO '06, ACM, New York, USA, 2006, pp. 609-610.

[8] B. Julstrom and B. Hinkemeyer. Starting From Scratch: Growing Longest Common Subsequences With Evolution. In Parallel Problem Solving from Nature - PPSN IX, vol. 4193 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 930-938.

[9] C. Kaner, Software Test Automation: A Real-World Problem, White paper, Los Altos Workshop on Software Testing 1-3, 1997-98.

[10] R. M. Karp and M. O. Rabin, Efficient Randomized Pattern-Matching Algorithms, IBM J. Res. Dev., vol. 31(2), March, 1987, pp. 249-260.

[11] D. E. Knuth, J. J. H. Morris, and V. R. Pratt, Fast Pattern Matching in Strings. SIAM Journal on Computing, vol. 6(2), 1977, pp. 323-350.

[12] R. W. Luk, H. V. Leong, T. S. Dillon, A. T. Chan, W. B. Croft, and J. Allan, A Survey in Indexing and Searching XML Documents, J. Am. Soc. Inf. Sci. Technol., vol. 53(6), May, 2002, pp. 415-437.

[13] D. Maier, The Complexity of Some Problems on Subsequences and Supersequences, Journal of the ACM 25, 1978, pp. 322-336.

[14] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," IEEE Transactions on Software Engineering, vol. 22, pp. 529–551, August 1996.

[15] K. G. Srinivasa, S. Sharath, K. R. Venugopal, and L. M. Patnaik, Gaxsearch: An XML Information Retrieval Mechanism Using Genetic Algorithms, In Australian Conference on Artificial Intelligence, 2005, pp. 435-444.

[16] J. Yang and R. R. Korfhage, Effects of Query Term Weights Modification in Annual Document Retrieval: A Study Based on a Genetic Algorithm, In Proceedings of the Second Symposium on Document Analysis and Information Retrieval, 1993, pp. 271-285.

[17] H. Zhu, X. Yang, B. Wang, and Y. Wang, Improving Text Search on Hybrid Data, In Web-Age Information Management, vol. 7419 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 192-203.

 ISBN: 978-1-61208-304-9