Information and Software Technology 58 (2015) 419–432

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Search based algorithms for test sequence generation in functional testing

Javier Ferrer [a,*], Peter M. Kruse [b], Francisco Chicano [a], Enrique Alba [a]

[a] Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, Spain
[b] Berner & Mattner Systemtechnik GmbH, Berlin, Germany

A R T I C L E   I N F O

Article history:
Received 6 May 2013
Received in revised form 26 July 2014
Accepted 26 July 2014
Available online 7 August 2014

Keywords:
Functional testing
Classification Tree Method
Test sequence generation
Search Based Software Engineering
Genetic Algorithm
Ant Colony Optimization

A B S T R A C T

Context: The generation of dynamic test sequences from a formal specification, complementing traditional testing methods in order to find errors in the source code.
Objective: In this paper we extend one specific combinatorial test approach, the Classification Tree Method (CTM), with transition information to generate test sequences. Although we use CTM, this extension is also possible for any combinatorial testing method.
Method: The generation of minimal test sequences that fulfill the demanded coverage criteria is an NP-hard problem. Therefore, search-based approaches are required to find such (near) optimal test sequences.
Results: The experimental analysis compares the search-based technique with a greedy algorithm on a set of 12 hierarchical concurrent models of programs extracted from the literature. Our proposed search-based approaches (GTSG and ACOts) are able to generate test sequences by finding the shortest valid path to achieve full class (state) and transition coverage.
Conclusion: The extended classification tree is useful for generating of test sequences. Moreover, the experimental analysis reveals that our search-based approaches are better than the greedy deterministic approach, especially in the most complex instances. All presented algorithms are actually integrated into a professional tool for functional testing.

1. Introduction

Software testing is a very important phase in the software development life cycle the goal of which is to ensure a certain level of software quality. The high economic impact of an inadequate software testing infrastructure was detailed in a survey [1]. In addition, it is estimated that half the time spent on software project development and more than half its cost, is devoted to testing the product [10]. The automation of test generation could reduce the cost of the whole project, this explains why both the software industry and academia are interested in automatic tools for testing. As the generation of adequate tests implies a big computational effort, search-based approaches are required to deal with this problem. Nowadays, automatic software testing is one of the most studied topics in the field of Search-Based Software Engineering (SBSE) [16,27].

Evolutionary Algorithms (EAs) have been the most popular search-based algorithms for generating test cases [27]. In fact, the term *evolutionary testing* is used to refer to this approach. In the paradigm of *structural testing* a lot of research has been carried out using EAs, but the use of search-based techniques in *functional testing* is less frequent [36], the main cause being the implicit nature of the specification, which is generally written in natural language.

Traditionally, the challenge has been to generate test suites to completely test the software. Complete testing is not feasible for arbitrarily large projects [21], so a good subset of all possible test cases has to be selected. Combinatorial Interaction Testing (CIT) [7] tries to address this problem. CIT approaches attempt to find a minimal test suite which fulfills the desired coverage. Generally, this task consists of generating, at least, all possible combinations of the parameters' values (this task is NP-hard [37]). The strength of the testing approach, t-strength, depends on the number (t) of parameters involved in the combinations (i.e., $t = 2$ for pairs, $t = 3$ for triples, etc.). Although combinatorial testing has been widely studied, we still find two main issues that have not been addressed by the traditional generation of test suites: the dependencies between individual test cases and the state of the software under test (SUT).

* Corresponding author. Tel.: +34 952133303.
E-mail addresses: ferrer@lcc.uma.es (J. Ferrer), peter.kruse@berner-mattner.com (P.M. Kruse), chicano@lcc.uma.es (F. Chicano), eat@lcc.uma.es (E. Alba).

http://dx.doi.org/10.1016/j.infsof.2014.07.014
0950-5849/© 2014 Elsevier B.V. All rights reserved.

Sometimes software is required to be in a particular state to test a given functionality. This is the case of most programs. Indeed, in very large software systems, the cost incurred to place the system in a certain state can be an issue. For example, testing the anti-lock braking system (ABS) of a car requires that the car reaches a certain speed before the system can be tested. So it makes sense to consider the generation of test sequences that allow us to test a particular functionality (acceleration of the car) while we change the state of the SUT (considering the dependency rules in the test cases) to test the next functionality (ABS). The implicit cost savings of using this technique is the reason why the generation of test sequences is relevant and deserves more research effort.

One CIT approach, the Classification Tree Method (CTM) [13] for functional testing, is used for test planning and test design. This method allows a systematic specification of the system under test and its corresponding test cases can be created automatically using CIT. Here, we extend the Classification Tree Method with transition information in order to be able to find the shortest test sequences.

We present a couple of metaheuristic approaches for computing optimal test sequences automatically. They are able to find near optimal solutions using a reasonable amount of resources [5]. We have compared the behavior of two metaheuristic techniques with an existing greedy algorithm [22]. The first proposed approach is a Genetic Algorithm (GA) called Genetic Test Sequence Generator (GTSG). We have improved a GTSG with the addition of a memory operator (MemO), which is based on the operator proposed by Alba et al. [3]. It is used to reduce the amount of resources needed to compute a solution.

The other proposed algorithm is an Ant Colony Optimization (ACO) [9]. Specifically, we propose a new technique based on an ACO algorithm that is able to deal with large construction graphs. It is able to find near-optimal solutions in separated areas of the search space for the Test Sequence Generation Problem (TSGP). It is called ACO for test sequence generation (ACOts). Both proposed metaheuristic approaches are used in our approach to generate test sequences to obtain full class and transition coverage of 12 different programs extracted from the literature. The main contributions of our approach are:

- We extend CTM in order to automatically generate test sequences. We formally define the Extended Classification Tree Method. Other combinatorial testing methods could be extended in the same way. The definition of an extended CTM could be done by a professional tool called CTE XL (see Fig. 3).
- We present an evolutionary test sequence generator for the CTM using a GA with a memory operator (MemO). In addition, we propose a new technique based on ACO (ACOts). These approaches can compute test sequences for full class and transition coverage without having to know the length of the sequences in advance.
- We perform an experimental analysis using 12 software models and comparing three different techniques.

The remainder of the paper is organized as follows. In Section 2 we present the background to the Classification Tree Method: how it is designed, how we have extended it and what is the adequacy criterion, and we briefly describe the CTE professional tool. Section 3 describes the Test Sequence Generation Problem and, then, it defines an extension of the classification tree in order to deal with test sequences. Section 4 presents our GTSG, ACOts, and outlines a deterministic greedy algorithm re-implemented for comparison purposes. Section 5 is devoted to presenting the benchmark of programs and analyzing the results of the three approaches. Section 7 surveys related work. Finally, in Section 8 some conclusions and future work are outlined.

2. The Classification Tree Method

The Classification Tree Method [13] is intended for systematic and traceable test case identification for functional testing over all testing levels (for example, component test or system test). It is based on the category partition method [31], which divides a test domain into disjoint classes representing important aspects of the test object. These classes can be seen as the states of the SUT. Applying the Classification Tree Method involves two steps: designing the classification tree and defining test cases. In addition, the extension of the Classification Tree Method and the coverage criteria are also described in this section.

2.1. Design of the classification tree

The classification tree is based on the functional specification of the test object. For each aspect of interest (called classification), the input domain is divided into disjoint subsets (called classes). Fig. 1 illustrates the concept of classification tree with a simple example for a video game. Two aspects of interest (*Game* and *Pause*) have been identified for the system under test. The classifications are refined into classes which represent the partitioning of the concrete input values. These partitions can also be further refined by introducing new low-level classifications and classes. In our example the refinement aspect *Playing* is identified for the class *runningGame* and it is divided into a further two classes *startup*, and *controlling*.

Given the classification tree, test cases can be defined by combining classes from different classifications. Since classifications only contain disjoint values, test cases cannot contain several classes of one classification. A test case for the running example is:

$$Game : \mathrm{runningGame}(Playing : \mathrm{startup}),\ Pause : running.$$

in which class *running* is selected from classification *Pause* and *runningGame* is selected from *Game*. Since class *runningGame* has an inner classification, *Playing*, we have to select a class from it, this class is *startup* in our case.

A test sequence is an ordered list of test cases or test steps which could be sequentially visited with the aim of completely testing the functionality of the whole system.

2.2. Extensions of the classification tree

The classification tree defined in the previous section can be used to design test cases in isolation. However, the test object can have operations related to transitions between classes in the classification tree and executing these transitions is the only way we can reach a given state (test case) of the object. Let us take our video game example and let us imagine that we need to execute some code when the user changes the state of the object from *starting game* to *running game*. These operations can be modeled by extending the Classification Tree Method with transitions between

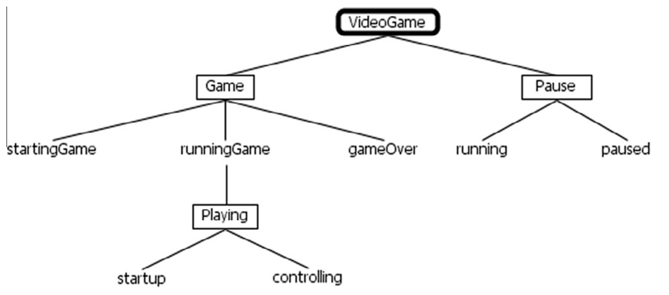

Fig. 1. Example of classification tree: *video game classification tree.*