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

ARTICLE INFO

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

ABSTRACT

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/

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:

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

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

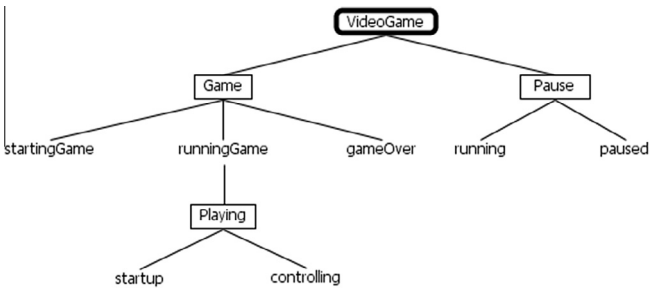

Fig. 1. Example of classification tree: *video game classification tree.*

classes (see Fig. 2). In a real-world example, these transitions come from the semantics of the software object. We also assume that each classification has a *default class* that we highlight in the graphical representation by underlining the class. This extension of the classification tree can be seen as a hierarchical concurrent state machine (HCSM) or statechart [15] where classes match states, and classifications match orthogonal regions.

When the transition information is available we are also interested in covering all the possible transitions in the system. In this case, sequences of test cases play a main role rather than the isolated test cases. In effect, an isolated test case does not describe which transitions were executed to get that test case and, thus, does not determine the transitions executed. For this reason, our goal in this work is to provide test suites composed of sequences of test cases that cover not only all the possible classes in the classification tree but also all the transitions using the minimal number of total test cases. We will give more details of the *Extended Classification Tree Method* (ECTM) in Section 3 and we will provide a formal definition and semantics in A.

2.3. Coverage criteria

In this paper we have chosen two coverage criteria: class and transition coverage. The class coverage criterion consists of covering all the classes of the classification tree with the generated test suite. The transition coverage requires covering all the transitions available between the classes of the ECTM. In our running example of Fig. 2, we have to cover eight classes for total class coverage (*VideoGame, startingGame, runningGame, startup, controlling, gameOver, running, and paused*), and five transitions to obtain full transition coverage ($\{startingGame \rightarrow runningGame,\ startup \rightarrow controlling,\ controlling \rightarrow gameOver,\ running \rightarrow paused,\ paused \rightarrow running\}$).

Similarly to the conventional test data generation, t-way sequences introduced by Kuhn et al. [23] can be mapped onto our coverage criteria: t-wise coverage for both classes and transitions. The 1-way sequence coverage of Kuhn et al. corresponds to 1-wise (or minimal) class coverage here. Each class is supposed to be contained at least once in the resulting test suite (or *result set* as Kuhn et al. call it). The 2-way sequence coverage of Kuhn et al. corresponds to our 1-wise (or minimal) transition coverage. All valid transitions between classes are supposed to be contained at least once in the result set. In conventional test case generation with the Classification Tree Method, there is no coverage criterion for transitions. Higher t-way (with $t > 2$) sequence coverage has not yet been included and requires further work.

2.4. Classification Tree Editor

The Classification Tree Editor [24] is a software tool supporting the Classification Tree Method (Fig. 3). It incorporates classification tree elements. Current versions of the CTE XL (professional) support automated test case generation and user-defined dependency rules; the valid transitions among classes could be defined by the user. However, the test sequence generation cannot be done automatically. In this paper we are going to deal with the automatic generation of test sequences. In the following section we describe the test sequence problem and we define how we interpret the extended classification tree.

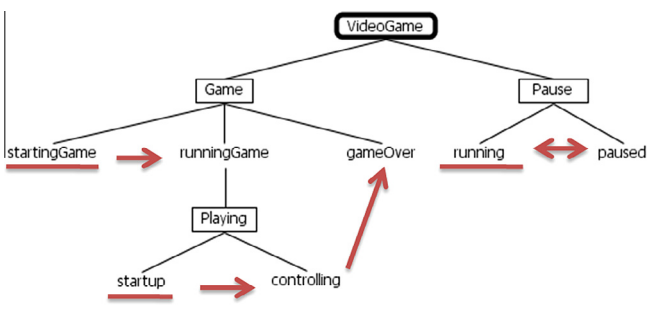

Fig. 2. Video game ECTM example.

3. Test Sequence Generation Problem

The problem of generating test sequences has received little attention in the existing literature, much less than the traditional generation of test data. As far as we know, this paper is the first in which the CTM has been extended to compute test sequences for functional testing. In addition to the constraints defined by the classification-classes hierarchy, in the Test Sequence Generation Problem (TSGP) we take dependency rules into account. These are constraints between single test steps, i.e., restrictions on the transitions between classes. Within each test sequence, dependency rules must not be violated.

Dealing with dependency rules is important since the testing of several states could be combined, resulting in shorter test sequences. In this way, we need fewer resources to test all functionality.

For example, testing a car at high speed implies using the accelerator pedal, but it is not possible to use the brake at the same time, so after it is necessary to test the brake. We could plan a sequence of test cases to check several functionalities instead of one. We could reduce the cost by testing the functionalities in a sequence. Since it would be more costly to test one functionality, then putting the system into an initial state to test the next functionality, than testing all the functionalities sequentially. In addition, it is desirable that the set of generated test sequences as a whole fulfills predefined coverage levels. So, it could be useful to generate a test suite with test sequences covering all possible classes or transitions between classes of the classification tree.

Our approach for test sequence generation is based on an idea proposed by Conrad [8], who suggests the interpretation of classification trees as parallel FSMs. However, we need to extend Conrad's approach to interpret refined classes of the classification tree. This concept is similar to the refinements of states in UML statecharts. Our approach can be seen as a statechart, because we have concurrent states and we have added hierarchies to the model. An example of a statechart can be seen in Fig. B.7, where we show the model of the Citizen watch by Harel [15]. Later, in the experimental section, we analyze this model in detail. We now describe in plain text the Test Sequence Generation Problem.

One *test case* for an ECTM is a set of classes that fulfills some rules. In particular, it is not possible to have two classes that belong to the same classification and if a refined class is in the test case then there must be one class for each classification in which the parent class is refined. In addition, if a class is in the test case, all the ascendant classes in the ECTM must be also included in the test case. For example, the set $Q = \{startingGame, running\}$ is a test case, but the set $Q = \{runningGame\}$ is not a test case because there is no class of the *Pause* and *Playing* classifications.

We can *transit* from one test case to another one by taking one of the transitions between classes. The test case we reach excludes the source class of the transition and includes the destination class of the transition. In order to fulfill the rules described for the test cases, some classes in the starting test case could also go out of the set and additional classes could enter the new test case. For example, if we take transition $startingGame \rightarrow runningGame$ from test case $Q_1 = \{startingGame, running\}$ in our video game example, we reach the test case $Q_2 = \{runningGame, startup, running\}$. We observe that class *startingGame* was removed from Q_1 and class

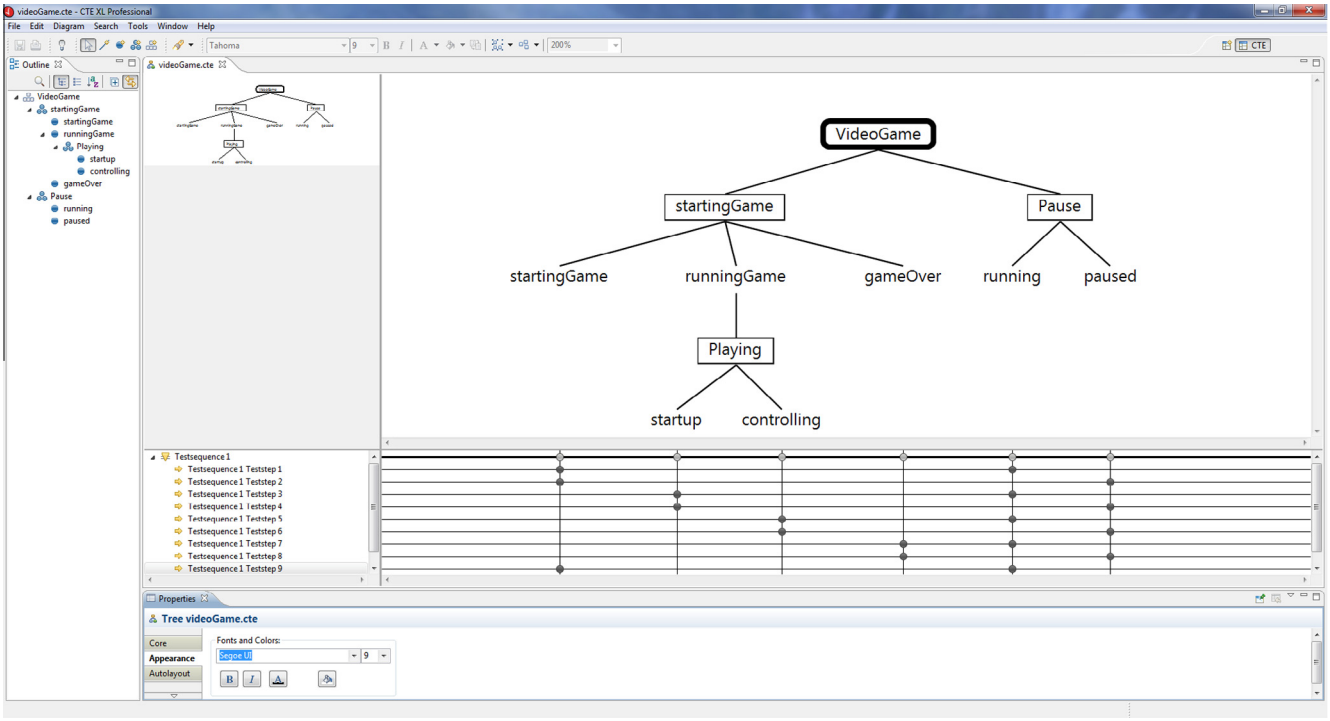

Fig. 3. CTE XL professional tool.

runningGame was added to Q_2, but we also need to add class *startup* because *runningGame* is a refined class. A *test sequence* is a list of test cases in which all except the first one are obtained by applying a transition from the previous one. A sequence of length three for our running example could be composed of test cases ($Q_1 = \{startingGame, running\}, Q_2 = \{runningGame, startup, running\}$ and $Q_3 = \{runningGame, startup, paused\}$).

If two different transitions can be used to transit in a given state and they affect different sets of classes it is possible to group them and consider one single step transition with the joint effect of both. In our example the transitions *startingGame* → *runningGame* and *running* → *paused* affect different sets of states since they belong to sibling classifications. Then, we can compose the transitions and build the test sequence of length two (Q_1, Q_3). This sequence covers the same transitions as the sequence (Q_1, Q_2, Q_3).

Given a test sequence we define the *class coverage* as the number of classes appearing in the test cases of the sequence divided by the total number of classes in the ECTM. We define the *transition coverage* as the number of transitions covered by the sequence divided by the total number of transitions. The problem we are interested in solving consists in finding a set of test sequences such that the coverage (class or transition, one each time) is maximized. For a more precise and formal definition of the concepts presented in this section the reader should refer to A.

4. Nature inspired algorithm for Test Sequence Generation Problem

In this section we describe three different approaches used to solve the TSGP. We first introduce an evolutionary approach, a Genetic Algorithm. Second, we describe our algorithmic proposal based on ACO for dealing with the TSGP. Finally, we briefly describe a state of the art technique from the literature for comparison purposes. We would like to highlight that the size of the test cases that compose a test sequence can vary from one to another. This fact is due to the hierarchical structure of the model. One class could be refined in several sub-classes, then the length of the test cases would be different. Consequently, we have to deal with the dynamic size of test cases in the ECTM.

4.1. Genetic Test Sequence Generator

The Genetic Test Sequence Generator (GTSG) constructs an entire test suite taking into account the dependencies between test data in the generation of the sequence. GTSG is an algorithm that evolves a population of solutions in each iteration until a given coverage criterion is fulfilled. The algorithm tries to find the tests that maximize the coverage, then it sequentially adds them to the solution (test sequence).

In Algorithm 1 we show the main loop of our GTSG. As input parameters, the algorithm needs the ECTM model, the algorithm parameters such as population size (GTSG.popSize), mutation probability (GTSG.Pm), and the coverage criterion used (criterion). To start, the test suite is initialized with an empty list (line 2) and the coverage set (`Coverage`) is initialized with all classes or transitions depending on the criterion selected. In line 4, the set *init* is initialized with the initial test case Q_{ini} (for a definition of Q_{ini} see A). In each iteration of the external loop, also called optimization step, (lines 5–22) the algorithm creates a random initial population of individuals (line 7). The first time the loop is entered it sets the initial test case of the sequence, in the subsequent iterations the initial test case is the last one stored in the Memory Operator which we describe in Section 4.1.1. Then, the GTSG enters an inner loop which applies the traditional steps of a generational evolutionary algorithm without recombination (lines 8–18). That is, some individuals (solutions) are selected from the population $P(t)$, they are mutated and evaluated, and they are finally inserted in the offspring population A.

Algorithm 1. Pseudocode of GTSG.

```
1: proc Input: (ECTM, GTSG, criterion)   // Inputs for 'GTSG'
2: TS ← ∅ // Empty the test suite list
3: Coverage ← initialize(criterion)// Initialize the coverage
   structure with classes or transitions
4: init ← {Q_ini} // Initial classes are the first test case of the
   sequence
5: while not empty(Coverage) do
6:    t ← 0
7:    P(t) ← create_population(init) // P = population
8:    while evals < totalEvals do
9:       A ← ∅ // A = auxiliary population
10:      for i = 1 to (GTSG.popSize/2) do
11:         parents ← selection(P(t))
12:         offspring ← mutation(GTSG.Pm, parents)
13:         evaluate_fitness(offspring)
14:         insert(offspring, A)
15:      end for
16:      P(t + 1) ← replace(A, P(t))
17:      t ← t + 1
18:   end while//internal loop
19:   TS ← addToList(best_sequence(P(t)))
20:   Coverage ← remove(best_sequence(P(t)))
21:   init ← MemoryOperator(best_sequence(P(t)))
22: end while   //optimization step
23: end_proc
```

In this particular algorithm the representation of a solution *sol* (test sequence) is a vector of integers of length *l*. We determine the length of the chromosome as a parameter of the memory operator (see next subsection).

$sol = [I_1, I_2, I_3, \ldots, I_l]$.

The outgoing transitions from a class of the current test case can be enumerated, thus each number (I_i) can be seen as the next transition chosen from the actual class to the next one.

The evaluation of a solution is done by sequentially taking every single transition (class to class) of the solution and generating a sequence of test data with a particular coverage. The evaluation function selects one leaf class (from left to right) and one gene in the solution is consumed to select the next transition t_i. Then, t_i is added to the set of selected transitions, $T\prime$. In order to transit from one test case to another, the evaluation function consumes, at most, as many genes as the number of leaf classes present in the source test case. We may need to consume a variable number of integer numbers of the solution to transit to the next test case. It depends on the source test case. We use the following expression to select the next transition:

$$t_i = I_i \bmod |Transitions(c)|. \quad (1)$$

where I_i is the i-th component of the Solution and $Transitions(c)$ is the list of possible outgoing transitions from class c.

For example, if the evaluation function is considering class c_i and that class has 4 outgoing transitions, we consume the next gene (integer), e.g. 6, in the solution to determine the next transition. In this example, we take the second possible transition ($t_i = 6 \bmod 4 = 2$).

The fitness value of a solution is the class or transition coverage, Eqs. (A.3) and (A.4), obtained by the solution when all genes have been consumed in the evaluation. In this algorithm we wish to maximize the fitness function given by Eq. (A.3) for Class Coverage and Eq. (A.4) for Transition Coverage.

The objective of the selection operator is to select several individuals from the population to which the other operators will be applied. The recombination operator is not used because the exchange of genes between two individuals could generate sequences of meaningless transitions. Since we interpret each gene in the chromosome as the transition to take from among all those possible, the interpretation of each number depends on the previously consumed numbers. Let us explain this issue in detail.

Let $I_1 = \{1,1,1,2,1,1\}$ and $I_2 = \{1,2,1,2,1,2\}$ be two individuals, and let us assume that after the application of the one-point crossover to them we obtain $I'_1 = \{1,1,1,2,1,2\}$ and $I'_2 = \{1,2,1,2,1,1\}$. The four first test cases in the sequences of I_1 and I'_1 are the same, since they have in common the first three transitions. Solutions I_2 and I'_2 also share the first four test cases. However, the fourth test case in I_2 is different from the fourth test case in I_1 (and I'_1). As a consequence, the last test cases of the sequence represented by I'_1 have nothing to do with the last test cases in I_2. Said in another way, the three last transitions, 2, 1, and 2, of I_2 have a completely different meaning when they appear in I'_1 because they are applied to a different test case. If these transitions in I_2 were appropriate because they traversed uncovered transitions and states, in I'_1 could not be the case. This fact is contrary to the philosophy behind the recombination operator, which tries to combine together features of the parent solutions. In summary, the natural recombination operator using this representation is quite disruptive and for this reason we do not use it.

Regarding the mutation operator, it iterates over all the components in the solution vector uniformly changing their value by ± 1. It linearly increases the probability to mutate a component in order to give a low probability to the first components of the chromosome, and a larger probability to the genes at the end of the chromosome. We aim to maintain the first part of the individual with fewer changes because a change in a gene could affect the rest of the sequence. We increase the probability from p_{m1} to p_{m2}. So here, $p_{m1} = 0.05$ and $p_{m2} = 0.25$.

In line 16, the best individuals of $P(t)$ and A are kept for the next generation $P(t+1)$. The internal loop is executed until a maximum number of evaluations is reached. Then, the best individual (partial sequence) found is added to the test suite list (line 19) and the `Coverage` set is updated by removing the classes or transitions which are going to be covered by the new best partial sequence. (line 20). Then, the MemO stores the last test case of the best sequence to be the initial test case for the next generation (line 21). Finally, the external loop starts again with a new population until there is no class or transition left in the `Coverage` set or the algorithm reach a predefined number of evaluations.

4.1.1. Memory Operator

In the aforementioned GTSG, as the population evolves, the first transitions in the individual tend to stabilize, but the algorithm still has to evaluate them at each generation. We propose saving the resulting stable first transitions in a memory slot to use them as the starting point for following optimization steps.

We use the memory operator (MemO) to allow the algorithm to search in stages. This operator was first proposed by Alba et al. [3] in the context of software verification. The algorithm can optimize the whole sequence of numbers (transitions) in stages, step by step, at the same time saving the memory required to evaluate complete individuals. Instead, we only have to evaluate a shorter sequence in each individual evaluation. This operator is based on the so-called missionary technique used in [2] for reaching deep graph regions in an ACO.

The advantages are obvious: less memory and time are required to evaluate an individual, and thus the path can maintain a

constant growth without requiring more time and memory. There are, of course, disadvantages. In particular, part of the search space is discarded and that part might in fact contain a good solution, but this is common in any non-exhaustive search algorithm.

We use the memory operator as follows: the GTSG is executed using a relatively small chromosome length (in this approach we use a chromosome length of 20 integers). After a predetermined number of evaluations (100,000), the memory operator selects the best individual and stores its transitions to use them as the starting points for the next optimization steps. The MemO could store more than one individual as the starting point for the next generation, but in accordance with previous experimentation performed in the early stages of this paper and the small population we used, the best choice is to select only the best individual. All the other transitions are removed from the memory.

4.1.2. Parameter settings

A possible threat to internal validity is that we have experimented with only one set of algorithms' parameters. Nevertheless, we have performed a previous experiment in order to select the best parameters for the GTSG algorithm. We have tried all combinations of values shown in Table 1. The parameters used in the final experimentation are the ones highlighted in bold in Table 1.

4.2. ACO Test Sequence

Our ACO Test Sequence (ACOts) algorithm is an adaptation of the ACOhg algorithm proposed by Alba and Chicano [2] that can deal with the construction of huge graphs of unknown size. This new model was proposed for applying an ACO-like algorithm to the problem of searching for counterexamples of safety properties in very large concurrent models. We have adapted the algorithm with the intention of solving the TSGP.

The ACO metaheuristic [9] is a global optimization algorithm inspired by the foraging behavior of real ants. The main idea consists of simulating the ants behavior in a graph, called a *construction graph*, in order to search for the shortest path from an initial set of nodes to the objective ones. The cooperation between the different simulated ants is a key factor in the search which is performed indirectly by means of *pheromone trails*, which is a model of the chemicals real ants use for their communication. The main procedures of an ACO algorithm are the *construction phase* and the *pheromone update*. These two procedures are scheduled during the execution of ACO until a given stopping criterion is fulfilled. In the construction phase, each artificial ant follows a path in the construction graph. In the pheromone update, the pheromone trails of the arcs are modified.

In short, two main differences between ACOts and the original ACO [9] model are as follows. First, the traditional ACO searches for the shortest path from an initial set of nodes to the objective ones. Since our objective in TSGP is to cover all classes or transitions, so we are also interested in visiting all classes and using all possible transitions between the first test case and the final test case. Second, ACOts cannot define final classes or test cases, the algorithm adds new test cases until the coverage criterion is fulfilled. In Algorithm 2 we present the pseudocode of ACOts.

Table 1
Parameters setting for GTSG. The parameter's values used in the experimentation are highlighted in bold.

Parameter	Value
Population size	**4**, 8, 10
Crossover	**No**, Yes (1.0, 0.9, 0.8)
Mutation prob.	0.05, 0.1, 0.2, **Dynamic (0.05–0.25)**
Memory operator	No, **Yes**
Memory slots	**1**, 2, 5
Chromosome length	10, **20**, 50, 100

Algorithm 2. Pseudocode of the ACOts algorithm.

```
1: proc Input: (ACOts)    //Algorithm parameters in 'ACOts'
2: init ← {Q_ini}; // Initial classes are the first test case of the
   sequence
3: τ ← initialize_pheromone();
4: step ← 1;
5: while step ⩽maxsteps∧ not empty Coverage(criterion)do
6:   for k = 1 to colsize do
7:     a^k ← ∅;
8:     while |a^k| ⩽ λ_ant ∧ T(a^k_*) − a^k ≠ ∅ do
9:       node ← select_successor(a^k_*, T(a^k_*), τ, η);
10:      a^k ← a^k + node;
11:    end while
12:  end for
13:  τ ← pheromone_evaporation(τ, ρ);
14: τ ← pheromone_update(τ, a^best);
15: step ← step + 1;
16: end while
17: compactSolution(a^best)
18: end_proc
```

In what follows we describe the algorithm, but prior to that we clarify some issues related to the notation used in Algorithm 2. In the pseudocode, the path traversed by the k-th artificial ant is denoted with a^k. We use $|a^k|$ to refer to the length of the path, the jth node of the path is denoted with a^k_j, and a^k_* is the last node of the path. Each node can be seen as a complete test case, the neighbors of a node are obtained by applying one single transition to the actual test case. We use the operator $+$ to refer to the concatenation of two paths. The set *init* is initialized with the initial test case Q_{ini} (for a definition of Q_{ini} see A).

The algorithm works as follows. First, the variables are initialized (lines 2–4). All the pheromone trails are initialized with the same value: a random number between 0.1 and 10. In the *init* set, a starting path with only the initial test case Q_{ini} is inserted (line 1). Therefore, all the ants begin the construction of their path at Q_{ini}.

After the initialization, the algorithm enters a loop that is executed until a given maximum number of steps have been performed or an ant reaches full coverage (line 5), depending on the coverage criterion. The boolean "Coverage" function returns the structure of coverage (set of classes or transitions) depending on the coverage criterion. For the construction of the path, the ants enter a loop (lines 8–11). In line 8, we use the expression $T(a^k_*) - a^k$ to refer to the elements of $T(a^k_*)$ that are not in the sequence a^k. That is, in that expression we interpret a^k as a set of nodes. In the loop each ant k stochastically selects the next node (line 9) according to the pheromone (τ_{ij}) and the heuristic value (η_{ij}) associated with each arc (i,j). In particular, if the last node of the k-th ant path is $i = a^k_*$, then the ant selects the next node $j \in T(i)$. $T(i)$ contains all possible transitions from all the classes in the current test case. Formally: $T(i) = T \cap (i \times C)$. Then, the next node is selected with probability

$$p^k_{ij} = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{s \in T(i)} [\tau_{is}]^\alpha [\eta_{is}]^\beta}, \quad \text{for } j \in T(i), \tag{2}$$

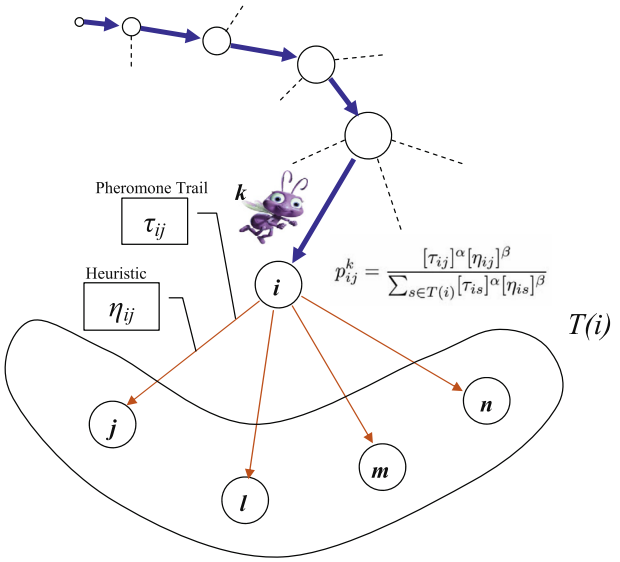

Fig. 4. An ant during the construction phase.

where α and β are two parameters of the algorithm determining the relative influence of the pheromone trail and the heuristic value on the path construction, respectively (see Fig. 4). According to the previous expression, artificial ants prefer paths with a higher concentration of pheromone, like real ants in the real world. When an ant has to select a node, the last node of the current ant path is expanded. Then the ant selects one successor node and the remaining ones are removed from the memory. This way, the amount of memory required in the path construction is small.

The heuristic function η depends on each arc of the construction graph and is defined in the context of ACO algorithms. It is a non-negative function used by ACO algorithms for guiding the search. The higher the value of η_{ij}, the higher the probability of selecting arc (i,j) during the construction phase of the ants. We use the same heuristic rate algorithm based on coverage of the greedy deterministic algorithm (Section 4.3) that will be presented in the following section.

The whole construction phase is iterated until the ant reaches the maximum length λ_{ant}, or it fulfills the coverage criterion. When all the ants have built their paths, a pheromone update phase is performed. First, all the pheromone trails are reduced, simulating the real world evaporation of pheromone trails, according to the expression $\tau_{ij} \leftarrow (1-\rho)\tau_{ij}$ (line 19), where ρ is the *pheromone evaporation rate* and it holds that $0 < \rho \leq 1$. Then, the pheromone trails associated with the arcs traversed by the best-so-far ant (a^{best}) are increased (line 14) using the expression shown in Eq. (3).

$$\tau_{ij} \leftarrow \tau_{ij} + \frac{1}{f(a^{best})}, \quad \forall (i,j) \in a^{best} \tag{3}$$

where $f(a^{best})$ is the percentage of coverage of the best-so-far ant.

This way, the best path found is awarded with an extra amount of pheromone and the ants will follow that path with higher probability in the next step, as in the real world. Once the termination condition has been fulfilled, the algorithm applies a compact function in order to minimize the steps of the a^{best}, resulting in the minimum number of different test cases. The compaction is as follows: since we only apply single transitions between classes, we can apply several transitions at the same time provided that the source class of the transitions is not the same or it is not an ascendant or descendant of the source class of any already selected transition. Then, we compact some of the single transitions in a complete transition that save some test cases in the resulting test suite. Continuing with the example shown in Fig. 2, if the actual test case is $Q_1 = \{controlling, running\}$, the following selected transitions are $controlling \rightarrow gameOver$ and $running \rightarrow paused$. Then, we can compact the two transitions in a complete transition to obtain directly $Q_2 = \{gameOver, paused\}$ in only one test step.

Table 2
Parameters setting for ACOts. The parameter's values used in the experimentation are highlighted in bold.

Parameter	Value
α	**1**, 2, 5
β	1, **2**, 5
ρ	0.1, **0.5**, 0.9
Maxsteps	10, 20, 50, **100**
ColSize	2, 5, **10**

4.2.1. Parameter settings

As in the case of CTSG, we did a previous experimental analysis to select the best parameters for ACOts. We have tried all combinations of values shown in Table 2. The parameters used in the final experimentation are the ones highlighted in bold in Table 1. The λ_{ant} parameter is set to 400 in order to allow large enough paths for finding the objective.

4.3. Greedy deterministic approach

This subsection describes an existing greedy deterministic approach, first introduced in [22] that will be used here for validation of our results against a consolidated technique in this problem. This approach uses a multi-agent system with two kinds of agents to traverse the classification tree: the walker agent and the coverage agent. Both agents will cooperatively traverse the ECTM.

4.3.1. Walker agents

Travelling is done in such a way that only valid paths are taken and that all traversed paths together result in the desired coverage. A full description of the algorithm has been given in [22], so we will only outline it here.

For any classification in the classification tree, a walker agent is introduced at the initial class. The initial test case is interpreted as a test step and taken into account for coverage calculation (e.g. class coverage, transition coverage). All walker agents are then moved one after another. The path of movement is calculated by coverage agents. When all agents have been considered once, the actual position of all agents is again interpreted as a test step, and is taken into account for coverage calculation, then added to the resulting test sequence. This is repeated until the desired coverage level has been reached. When there are no more valid paths to take, walker agents are stuck. In this case, the whole ECTM is reset to its initial state and a new, additional sequence is created. When the algorithm has finished a test suite with all test steps is returned.

4.3.2. Coverage agents

The Heuristic Rate algorithm is run by the aforementioned coverage agent. This agent guides the main algorithm to achieve full class and/or transition coverage.

Algorithm 3. Pseudocode of the Heuristic Rate algorithm.

```
1: proc Input:candidate (Class or Transition)
2: if classcoverage && selfTransition then
3:    return 0
4: end if
5: weight = 1.0; rating = 0
6: queue ← ϕ
7: queue += (candidate, weight)
8: while !queue.empty() do
9:    (item, weight)= queue.poll()
10:   if (item==candidate && rating > 0) then
11:     rating+=100; continue
12:   end if
13:   if (ratedItems contains item) then
14:     continue
15:   end if
16:   ratedItems+=item
17:   if (targetNodes contains item) then
18:     rating+=10∗weight
19:   end if
20:   if (item has (outgoing transitions || childnodes ||
    classifications)) then
21:     weight = weight∗0.95;
22:   end if
23:   for all (item has (outgoing transitions && childnodes &&
    classifications) of item) do
24:     queue+=(item, weight)
25:   end for
26: end while
27: return rating
28: end_proc
```

The Heuristic Rate algorithm is outlined in Algorithm 3. Its main goal is to rate the candidate transitions or classes in order to decide which is going to add more coverage to the current solution. The heuristic algorithm gets a candidate class or transition as input. For class coverage, self transitions are ignored and then zero is returned. A self-transition does not increase class coverage because the origin and the end of the transition is the same class. Otherwise, it then adds this candidate to a queue together with a weight factor, with an initial weight factor of one. A weight factor is needed to give more weight to the closest uncovered classes than those farthest away. The initial rating is set to zero. The candidate is added to the list of rated items. Then, while the queue is not empty the algorithm polls (FIFO) the next class and weight factor from the queue. If the polled node is the original candidate and if the rating is larger than zero, the algorithm has found a loop path with new items. This loop path is weighed by adding the value of 100 to the rating because we found a promising candidate. In this case or when the current item is on the list of rated items, the while loop passes to its next cycle. Otherwise this node is added to the list of rated items. If the node is on the list of target classes (it has not been used in any test step before), the algorithm adds 10 times the weight factor to the result rating. Then, if there are outgoing transitions, child nodes or classifications, the weight factor is multiplied with a punishment value. Target classes of outgoing transitions and child classes are then added to the queue together with the new weight factor. When the queue is empty the rating is returned.

5. Experiments

This section describes the experiments performed on a benchmark of programs. In the first subsection we present the benchmark of programs that we use in the experimental section. Then, in the second subsection we analyze the results of the comparison between the algorithms.

5.1. Experimental benchmark

For the experiments we use a benchmark with 12 different models of programs/artifacts. We use a Keyboard instance [28], a Microwave [25], an Autoradio [18], and Harel's Citizen watch [15] which is relevant in the literature. From the IBM Rhapsody instances, we took the Coffee Machine, the Communication example, the Elevator, and the Tetris game [20]. In Matlab Simulink Stateflow, we found Mealy Moore, Fuel Control, Transmission, and Aircraft [26]. Even though the details of the case studies are given in Table 3, we highlight here that most instances are hierarchical and concurrent. This means that we are going to deal with test cases of different lengths. In other words, there are test cases of different lengths in the same sequence.

In Table 3 the second and third columns list some statistics of the resulting artifacts. Both the number of classes and number of transitions are given. The fourth and fifth column list the results for conventional test case generation computed by the CTE tool with the greedy algorithm for minimal and complete combination. Numbers indicate the size of the generated test suite.

5.2. Experimental settings

ACOts and GTSG are non-deterministic algorithms, so we performed 30 independent runs per program/coverage criterion for a meaningful statistical analysis. In order to check whether the differences between the algorithms are statistically significant or just a matter of chance, we applied the Wilcoxon rank-sum [32] test and highlight in the tables, the differences that are statistically significant. We set a confidence level of 99.9% (p-value under 0.001) for the entire comparison (both metaheuristics acting on a program/coverage). We have marked a result in dark grey when it is the best and in light grey when it is the second best in performance. When the result of one algorithm is significantly better than the result of another algorithm (typically the one whose results is farthest), we have added an asterisk. Two asterisks are added if the algorithm is significantly better than the other two algorithms. In addition, with the aim of properly interpreting the results of statistical tests, it is always advisable to report effect size measures. For that purpose, we have also used the non-parametric effect size measure $\widehat{A}_{12}$ statistic proposed by Vargha and Delaney [35]. Effect size provides information about the magnitude of an effect, which can be useful in determining whether it is of practical significance or not.

All the executions were run in a cluster of 16 machines with Intel Core2 Quad processors Q9400 (4 cores per processor) at 2.66 GHz and 4 GB memory running Ubuntu 12.04.1 LTS and managed by the HT Condor 7.8.4 cluster manager.

Table 3
General characteristics of the benchmark of programs.

Name	Classes	Transitions	Minimal	Complete
Keyboard [28]	5	8	2	4
Microwave [25]	19	23	7	56
Autoradio [18]	20	35	11	66
Citizen [15]	62	74	31	3121
Coffee Machine	21	28	9	81
Communication	10	12	7	7
Elevator	13	18	5	80
Tetris	11	18	10	10
Mealy Moore	5	11	5	5
Fuel Control	5	27	5	600
Transmission	7	12	4	12
Aircraft	24	20	5	625

Table 4
Results for test sequence generation for class coverage.

Name	GTSG	ACOts	Greedy
Keyboard [28]	2	2	2
Microwave [25]	8*	8*	9
Autoradio [18]	13.30*	14	13*
Citizen [15]	39.47*	36**	47
Coffee Machine	9	9	9
Communication	7	7	7
Elevator	6	6	6
Tetris	12*	12*	15
Mealy Moore	5	5	5
Fuel Control	5	5	5
Transmission	4	4	4
Aircraft	4 *(86.2%)*	4 *(86.2%)*	4 *(86.2%)*

Table 5
Vargha and Delaney's statistical test results ($\widehat{A}_{12}$) for class coverage. *A* represents algorithms in rows and *B* represents algorithms in columns.

	GTSG	ACOts	Greedy
GTSG	–	0.5139	0.3889
ACOts	0.4861	–	0.4167
Greedy	0.6111	0.5833	–

Let us explain the notation used in the table of results. A single number n indicates the size of the unique test sequence: it is the number of generated test steps n needed for 100% coverage. In the case of the metaheuristic algorithms, we provide the mean over the 30 executions. A number n followed by a percentage value $(p\%)$ indicates the number of generated test steps n together with a coverage level $p\%$ below 100%. When the number n is followed by another number (m), the first number n indicates the total number of test steps while the second number m in parentheses indicates the number of sequences needed. We have implemented a re-boot mechanism in all the algorithms in case they reach a class with no exit transition.

5.3. Experimental analysis

In this section, we analyze the behavior of the proposed approaches with the aim of analyzing the computed best solutions and highlighting the algorithm that behaves the best. The main results of the executions of the algorithms for class coverage and transition coverage can be seen in Tables 4 and 6, respectively.

For *class coverage*, 100% coverage was reached for 11 out of 12 programs. Achieving full coverage is the main objective for test case generation. The Aircraft program was the only one resulting in below 100%, having an 86.2% coverage, as there are unreachable or orphaned classes in the model. In all 12 programs, the highest possible class coverage was reached in a single test sequence, which is a desirable result. Regarding class coverage, differences appear in four programs (Microwave, Autoradio, Citizen, and Tetris). The greedy approach obtains better results in the Autoradio program, where the difference with GTSG is not significant. For the other three programs the metaheuristic algorithms achieve total coverage using fewer test steps. For instance, both metaheuristic algorithms reduced the test suite size by more than 20% for the Tetris program.

Let us analyze the Citizen program for class coverage. The analysis of this program is especially interesting because this is the most complex program. Furthermore, the differences between the algorithms are the largest. ACOts obtains the best results in this program. ACOts reduces the test suite size by more than 23% with respect to the Greedy algorithm, moreover it is 9% better than GTSG. In addition, GTSG is 15% better than the Greedy algorithm. The ACOts approach is more effective and accurate for the largest model used in this study.

In light of these results and with the intention of determining whether the results are of practical significance or not, we analyze the $\widehat{A}_{12}$ statistic as follows: given a performance measure M, $\widehat{A}_{12}$ measures the probability that running algorithm A yields higher M values than running another algorithm B. If these two algorithms are equivalent, then $\widehat{A}_{12} = 0.5$. If $\widehat{A}_{12} = 0.3$ entails one would obtain higher values for M with algorithm A, 30% of the times. In this regard, A represents algorithms in rows and B represents algorithms in columns. In Table 5 we summarize the average of the $\widehat{A}_{12}$ statistic values for class coverage and all programs. The differences between algorithms are not very large due to we have selected small, medium, and large programs. Consequently, it is very difficult to obtain large differences in small and medium

Table 7
Vargha and Delaney's statistical test results ($\widehat{A}_{12}$) for transition coverage. *A* represents algorithms in rows and *B* represents algorithms in columns.

	GTSG	ACOts	Greedy
GTSG	–	0.5125	0.4670
ACOts	0.4875	–	0.4545
Greedy	0.5329	0.5455	–

Table 6
Results for test sequence generation for transition coverage.

Name	GTSG	ACOts	Greedy
Keyboard [28]	5	5	5
Microwave [25]	17	17	17
Autoradio [18]	36.30	36	36
Citizen [15]	75.27* *(99.9%)*	64.17**	51 *(92.7%)*
Coffee Machine	19	19	18**
Communication	16*	16*	17
Elevator	9	9	9
Tetris	31	31	31
Mealy Moore	24	24	24
Fuel Control	11*	11*	12
Transmission	9	9	9
Aircraft	7 (2)	7 (2)	7 (2)

models. Numerically, the results of the ACOts are going to be better than the ones provided by GTSG and Greedy in 51.39% and 58.33% times, respectively. In addition, the results of GTSG are going to be better than the Greedy ones in 61.11% times, which is a big difference.

For *transition coverage* (Table 6), only ACOts is able to obtain 100% coverage in all the programs. The other two algorithms fail to obtain total coverage in the one program (Citizen). In 11 of the 12 programs, the result only consisted of one test sequence, while in the Aircraft program two sequences were generated. We have implemented a re-boot mechanism in all the algorithms in case they reach a class with no exit transition, this is the reason why two sequences are needed to reach total coverage in the Aircraft program. The differences appear in five programs (Autoradio, Citizen, Coffee, Communication, and Fuel Control). In this case the Greedy algorithm is only better than the others in the *Coffee Machine* program, the Greedy algorithm reduces the test suite size, in this program, in one test case compared to the metaheuristic approaches. The existing differences are low in most cases except in the Citizen program where ACOts is clearly the best. It is the only algorithm that always achieves 100% transition coverage for all the programs. In the Citizen program the Greedy algorithm does not achieve full transition coverage while GTSG obtains total coverage in most executions. ACOts is better than GTSG in coverage and test suite size. ACOts is able to reduce the test suite size by 14.7% (with respect to GTSG).

Table 7 shows the $\widehat{A}_{12}$ statistical results for measuring the effect size for transition coverage. We have considered all programs, with the exception of the Citizen program where the results are not comparable. This fact is because neither GTSG nor the Greedy algorithm are able to reach full transition coverage, so the test suite is shorter but it is quite worse in quality (coverage). Although we have not included the Citizen results, where the ACOts algorithm is clearly superior, ACOts is still better than GTSG and Greedy by 51.25% and 54.55%, respectively. Furthermore, GTSG obtains smaller test suites than Greedy by 53.29%. Regarding the solution quality (coverage level), the metaheuristic approaches (ACOts and GTSG) seem to be competitive. They are both capable of generating test sequences with maximum levels of coverage, and obtain better results than the Greedy algorithm with a high probability.

5.4. Test suite coverage versus test suite size

Another aspect that we must take into account is the increase in the test suite size with the coverage in order to obtain total coverage. This behavior requires a further analysis to evaluate

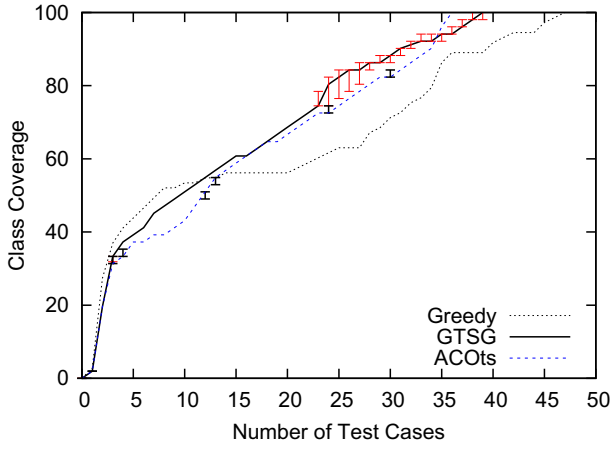

Fig. 5. Median solutions and interquartile range of ACOts, GTSG and Greedy algorithms for the Citizen example for Class Coverage. Coverage versus number of test cases in the solution.

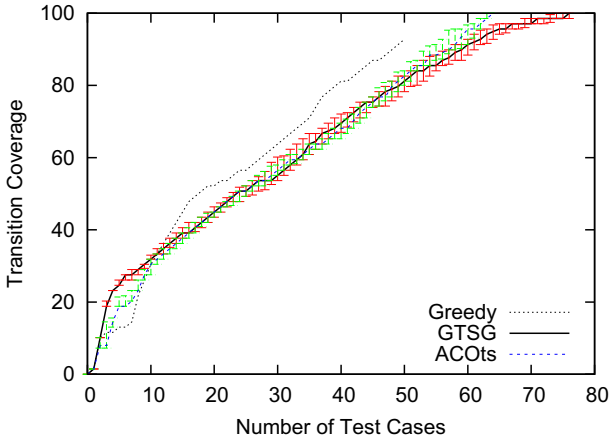

Fig. 6. Median solutions and interquartile range of ACOts, GTSG and Greedy algorithms for the Citizen example for Transition Coverage. Coverage versus number of test cases in the solution.

the tradeoff between coverage and test suite size because this is a key aspect when you are generating test suites [29]. We illustrate this tradeoff for the Citizen program in Fig. 5, and in Fig. 6 for class and transition coverage. In the figures, we show the deterministic solution of the Greedy algorithm and the median and interquartile range of the 30 executions of the metaheuristic algorithms in order to capture the average behavior of the approaches. We would like to stress that this analysis is performed on the solutions, already computed.

Let us start with the analysis of the solution where we want to cover all the classes (class coverage) of the Citizen program. In Fig. 5 we show that the obtained coverage is similar for the first test steps. The Greedy algorithm is slightly better with up to 54% coverage. Then, both metaheuristic algorithms continue adding coverage with the same ratio in contrast with the Greedy algorithm, which is worse in the middle stage of the sequence. GTSG obtains its maximum advantage when it achieves 80% coverage, while ACOts only achieves 72% with the same test steps (24). When only a few classes remain unvisited, ACOts is able to visit them in fewer test steps. Thus, it achieves full class coverage in only 36 test cases, three test cases less than GTSG and 11 test cases less than the Greedy algorithm. ACOts obtains total coverage with only 64 test cases in median, meanwhile GTSG has achieved 94.12% and the Greedy algorithm has achieved only 89.04% coverage with the same number of test cases.

In certain regions of the graph (Fig. 5) we observe that the same coverage is repeated in consecutive test steps. The reason is that not every class can be reached from any other class, but requires additional traversal of other covered classes and, therefore, additional test steps. We see this behavior, in particular, in the solution of the Greedy algorithm for the class coverage. This implies that we are not adding any coverage in these traversal test steps, so our algorithm should minimize them.

In Fig. 6 we show the median transition coverage and the interquartile range of the proposed algorithms achieved with each test case of all test sequences (average of 30 executions of non-deterministic algorithms). In this case, GTSG is better at the beginning because it first explores an area with a higher density of transitions (i.e., the algorithm does not have to visit an already visited node to reach a non-visited node). Besides, the Greedy algorithm obtains better coverage using the same number of test cases from 12 test cases onwards, but it is not able to achieve more than 92.7% coverage. Although the Greedy algorithm has achieved 11.59% more coverage than GTSG and 10.14% more than ACOts with 51 test cases, both metaheuristic algorithms are able to reach

full coverage. In this case, ACOts is better because it achieves full transition coverage in fewer test cases and it adds coverage in each test case, progressively. This great effort in reducing traversal test steps makes the algorithm reasonably predictable. This behavior is desirable because the obtained coverage is proportional to the test cases needed to reach certain levels of coverage.

For the goal of test suite minimization we have tried to optimize the test suite sizes while still achieving high levels of coverage. We have used GTSG and ACOts to search for the optimal solution but we need to evaluate the minimal test suite size using an exact approach, allowing us to know if we have reach the optimal one. Regarding computation times, we can say that the generation times for GTSG is less than 10 min in average, for ACOts is less than a minute in average, while the deterministic algorithm takes around 10 s in average. If we take into account the performance and the quality of the obtained results, it seems that ACOts is the best option, at least for the largest instances.

6. Threats to validity

A possible threat to internal validity is that we have not experimented with all possible configuration settings for the algorithms' parameters. However, these settings are in accordance with the common guidelines in the literature and our previous experience in testing problems. Parameter tuning can improve the performance of the algorithms, although default parameters often provide reasonable results [4].

We ran our experiments on an industrial case study to seek the best solution to minimize test suites for testing a product. To reduce external validity threats (i.e., our results might not be applicable to other empirical studies), we have used 12 case studies. The most probable conclusion, is that validity threats in experiments involving randomized algorithms, is due to random variations. To address this, we repeated the experiments 30 times to reduce the possibility that the results were obtained by chance. Furthermore, to determine the probability of yielding higher performance by different algorithms, we measured the effect size using $\widehat{A}_{12}$ statistic proposed by Vargha and Delaney [35]. We chose the $\widehat{A}_{12}$ statistic as it is appropriate for non-parametric effect size measure, which matches our situation. Meanwhile, we performed the Wilkoxon test to determine the statistical significance of the results.

7. Related work

It is not the first time that an ACO-like algorithm has been applied to a problem in the software engineering domain. Several papers [14] have been published showing promising results using ACO-like algorithms. These kinds of algorithms seem to be a good choice for dealing with test sequences.

Windish has applied search-based testing to Stateflow Statecharts [38]. In his work he dealt with hierarchical structures such as subsystems in order to reduce the complexity of the model. In the approach the optimization sequence consisted of only a small number of parameters to be used for the optimization engine and to be transformed into a simulation sequence by interpolation. However, in our work we use more complex instances of software objects like a microwave, a watch and a coffee machine. The technique used for test sequence generation in our work was introduced by Kuhn, Kacker and Lei: they generated event sequences for a given set of system events. Their approach is based on *t*-way sequences, which includes all *t*-events being tested in every possible *t*-way order [23].

In [30] a messy Genetic Algorithm (GA) is used to generate transition tours through Simulink Stateflow models. The authors identify two main challenges: trigger blocks containing timing constraints or counters and cyclic paths which may require several traversals before triggering a transition. A further problem is the a priori unknown length of the resulting tour. Stateflow models support hierarchies and concurrencies which they directly use to avoid sequentialization and therefore do not suffer from state explosion. They apply their approach to three well-known instances.

There has also been much work done on greedy algorithms for generating test data and test sequences. In particular, Gargantini and Riccobene [11] discuss automatic test sequence generation and coverage criteria for testing abstract state machines. Ural [34] describes four formal methods for generating test sequences based on a finite-state machine (FSM) description. The question to be answered by these test sequences is whether or not a given system implementation conforms to the FSM model of this system. Test sequences consisting of inputs and their expected outputs are derived from the FSM model of the system, after which the inputs can be fed into the real system's implementation. Finally, the outputs of the model and the implementation are compared.

Geist et al. [12] divide a test problem into aspects of interest to guide the search for test cases to interesting parts of the system, using temporal logic and Binary Decision Diagrams (BDDs) instead of traditional graph-algorithmic models. The target is transition coverage. All FSM transitions are stored in a BDD for performance reasons. Test cases are generated per transition. New test cases are evaluated for all included transitions and removed from the list of transitions to be covered. Their generation creates many test sequences of medium length, so they propose, in future work, to create longer test sequences. Heimdahl et al. [17] briefly survey a number of approaches in which test sequences are generated using model checking techniques. The idea is to use the counter-example generation feature of model checkers to produce relevant test sequences.

Techniques based on a formal specification of the software have also been studied. Burton et al. [6] present an approach which uses formal specification from statecharts and a testing heuristic to automatically generate test cases. For all transitions in the statechart a Z-representation is extracted. The Z-representation is then used to create an internal representation. A test sequence is then created for each state of the internal representation. There is no minimization of test sequences. To generate tests from Z specifications, the disjunctive normal form (DNF) method can be used, although it is prone to state explosion. Hierons et al. [19] propose the construction of a classification tree from the Z specification and use the resulting tree for test generation. There are several suggestions for constraint learning and efficient tree construction, although the main manual work of test case selection is left to the tester.

We have studied these previous techniques and we have tried to improve upon their weaknesses and integrate their strengths, in our work.

8. Conclusions

In this paper we have extended one CIT approach, the Classification Tree Method by test sequence generation. We have defined an entire model (ECTM) which both industry and academia could use to completely describe all aspects needed to generate sequences of tests for testing a program. Its benefits are clear, we can save costs and time executing all test steps sequentially because the previous test step puts the software in the adequate state to test the next functionality.

We have presented two different metaheuristic approaches to optimize the automatic generation of test sequences for the Classification Tree Method. The first is a GA with memory operator (GTSG), which is able to preserve the memory required to evaluate individuals, while also allowing the algorithm to compute a

solution faster than without the operator. The second is an ACO algorithm, concretely, we propose ACOts, a variation of the ACOhg implementation that is able to obtain good quality solutions, using little memory.

We have also compared our results with the ones of an existing greedy deterministic algorithm. We have used the algorithms to find test sequences for 12 different programs extracted from the literature. After analyzing the solutions obtained by the three approaches, we can conclude that the metaheuristic approaches are significantly better than the greedy deterministic approach for the largest model of program, specially the ACOts algorithm. The Greedy algorithm is only better than GTSG and ACOts in, respectively, 1 and 2 out of 8 scenarios where statistical differences exist. GSTG is statistically better than the Greedy algorithm in 4 out of 8 scenarios. Finally, ACOts is better than the Greedy algorithm in 6 out of 8 scenarios where statistical differences exist. Therefore ACOts is the best algorithm in the comparison. It has a good tradeoff between test suite size and coverage.

Further research will focus on dealing with t-wise coverage criteria with higher t (with $t \geqslant 2$) for test sequences. In other words, we need efficient algorithms able to compute pairwise class or transition coverage that might require an exponential growth of test sequences to fulfill a stricter coverage requirement. The pairwise coverage will add more confidence to the testing phase. Moreover, we need to collect more real scenarios for comparison purposes, this is absolutely necessary when you are adding functionality to a professional tool such as CTE XL. Finally, although we have obtained great results with the ACOts algorithm, we plan to propose a trajectory search-based algorithm such as Simulated Annealing that has obtained great results in Combinatorial Testing (Covering Arrays [33]) and might suit this problem.

Acknowledgements

This research is partially funded by the Spanish Ministry of Economy and Competitiveness and FEDER under contract TIN2011-28194 and fellowship BES-2012-055967. It is also partially funded by project 8.06/5.47.4142 in collaboration with the VSB-Tech. Univ. of Ostrava, Universidad de Málaga, Andalucía Tech. and EU Grant ICT-257574 (FITTEST project).

Appendix A. Formal definition of the Extended Classification Tree Method

In this section we formally define the extended model of the Classification Tree Method in order to describe all the aspects needed to generate sequences of tests for testing an artifact. The ECTM model can be totally defined by a tuple of four elements:

$$\text{ECTM} = (C, V, w, T), \tag{A.1}$$

where C is the set of Classes, V is the set of Classifications, w is a word of the language $L(G)$ generated by the grammar G defined next and T is the set of allowed transitions between the classes $T \subseteq C \times C$. We will use either the notation $c_s \rightarrow c_d$ or (c_s, c_d) to represent a transition between classes c_s and c_d. The grammar G is defined as:

$$G = (N, \Sigma, P, S) \tag{A.2}$$

where N is the set of nonterminal symbols: N = {Class, AtomicClass, RefinedClass, Classification}. Σ is the set of terminal symbols: $\Sigma = C \cup V \cup Punct$, where $Punct$ contains the squared brackets and comma. P is the following set of production rules:

$$\begin{array}{lcl}
S & \rightarrow & Class \\
Class & \rightarrow & AtomicClass | RefinedClass \\
AtomicClass & \rightarrow & \beta \quad \forall \beta \in C \\
RefinedClass & \rightarrow & \beta\ [Classification\ (, Classification)*] \quad \forall \beta \in C \\
Classification & \rightarrow & [\alpha, \gamma, [Class\ (, Class)*]] \quad \forall \alpha \in V,\ \forall \gamma \in C
\end{array}$$

where γ in the last rule represent the initial (default) class of classification α. S is the axiom of the grammar. The language $L(G)$ generated by the grammar represent all the possible trees that can be build using the same set of classes and classifications.

As an illustration, the ECTM model shown in Fig. 2 can be defined by the following quadruple:

$$ECTM_{Ex1} = (\{VideoGame, startingGame, runningGame, startup, controlling, gameOver, running, paused\}, \{Game, Playing, Pause\}, w, \{startingGame \rightarrow runningGame, startup \rightarrow controlling, controlling \rightarrow gameOver, running \rightarrow paused, paused \rightarrow running\})$$

where the word w for this example is:

```
w :=VideoGame [
  [Game, startingGame,
    [startingGame,
    runningGame [Playing, startup, [startup,
  controlling]],
    gameOver]],
  [Pause, running, [running, paused]]]
```

Let us define some relations between the elements e (classes and classifications) in the ECTM model. An element e_p is *parent* of e_d, if e_d belongs to one of the classifications or classes defined by e_p. If e_p is parent of e_d, then we say that e_d is a child of e_p. The *ascendant* relation is the transitive closure of the parent relation and the *descendant* relation is the transitive closure of the children relation. In our example, the class *runningGame* is the parent of the classification *Playing* and is also the ascendant of the classes *startup* and *controlling*. On the other hand, *Playing* is child of *runningGame*, meanwhile, the three elements (*startup*, *controlling* and *Playing*) are descendants of *runningGame*.

An element e_s is *sibling* of another element $e_{s'}$ if they have the same parent. For example the class *startingGame* is sibling of *runningGame* and *gameOver*. In addition, the classification *Game* is sibling of the classification *Pause* and vice versa. The initial class γ of a classification v is defined in the word w. Finally, the *root* class is the first element that appears in w and it does not have a parent in the tree (it only has descendants). From these relations we define all the related functions that, given an element, return a set of elements: $Parent(e), Ascendants(e), Children(e), Descendants(e), Siblings(e)$ and $InitialClass(v)$.

The transition set in an ECTM model can contain any transition except those connecting classes of sibling classifications. Formally, any ECTM model must fulfill:

$$\forall v_1, v_2 \in V, v_1 \in Siblings(v_2) \Rightarrow \forall c_1 \in Descendants(v_1) \cap C, \forall c_2 \in Descendants(v_2) \cap C, (c_1, c_2) \notin T.$$

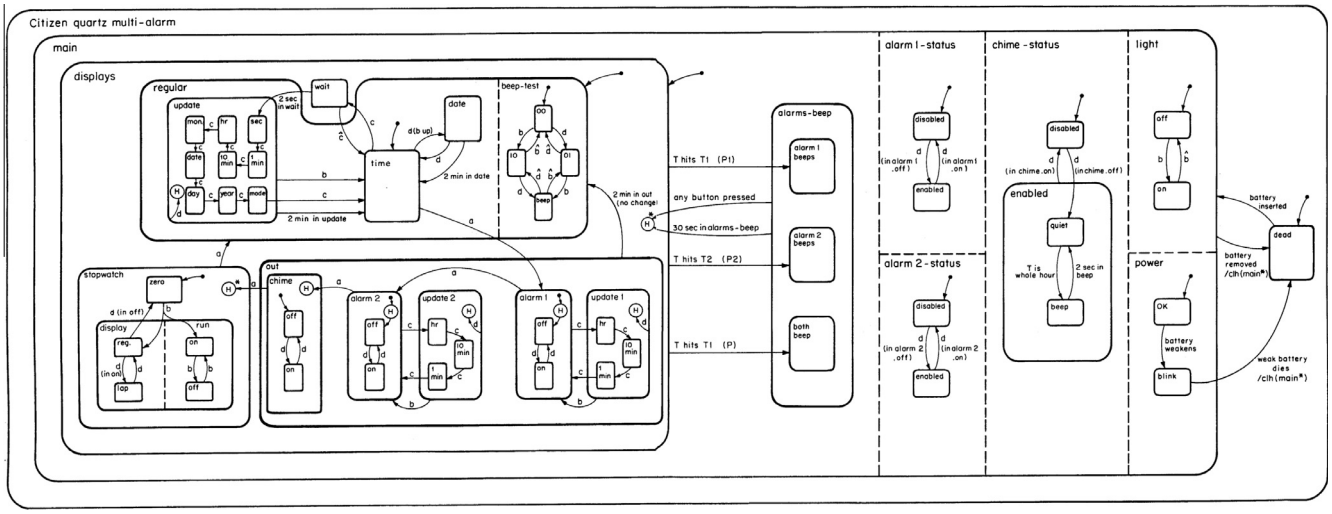

Fig. B.7. Harel's HCSM of the Citizen watch.

A valid test case for a particular ECTM model is a set of classes:

$$Q := \{c_1, c_2, \ldots, c_n\}$$

where the classes c_i must fulfill the following rules:

1. $\forall c \in Q \setminus root, Ascendants(c) \cap C \in Q$.
2. $\forall c \in Q, \forall s \in Children(c), \exists d \in Q, d \in Children(s)$.
3. $\forall c, b \in Q, c \neq b \Rightarrow b \notin Siblings(c)$.

Rule 1 says that if a class is in the test case then all the classes in which it is included (ascendant classes) must also be in the test case. Rule 2 requires that all the classifications under a class that is in Q must have a class in Q. Finally, Rule 3 prevents from having two classes of the same classification in the test case.

In order to build a sequence of test cases we must define how to navigate from a source test case Q_1 to a destination test case Q_2. The initial test case in a sequence, Q_{ini}, is composed by the initial classes of the children classifications under the root of the tree and all their ascendants. Given a transition $t = (c_s, c_d) \in T$, the general rule to transit from Q_1 to Q_2 is as follows. We must find the deepest common classification of c_s and c_d, say v_a. If there exists another common classification, then that classification must be an ascendant of the deepest one v_a. Once we have found v_a, we must remove from the source test case Q_1 all the classes under v_a, in other words, any class that is descendant of v_a. Next, we have to add c_d and its ascendants which are children of v_a, and add the initial classes of the classifications of these ascendants, except the siblings of c_d. If c_d is a refined class, then the initial classes of all classifications of c_d and their descendants are also added in order to build a valid test case. Let us formally define all this procedure.

Let Q_1 be the source test case, first we must remove from Q_1 the descendants of v_a:

$$Q' = Q_1 - Descendants(v_a)$$

where $\{c_s, c_d\} \subseteq Descendants(v_a)$ and does not exist $v_d \in Descendants(v_a)$ such that $\{c_s, c_d\} \subseteq Descendants(v_d)$. Then, we must add some classes to Q' in order to transit to the new test case Q_2. In order to do this, let us define the function $Incomplete(Q)$ as follows:

$$Incomplete(Q) = \{v \in V | Parent(v) \in Q \wedge \forall c' \in Children(v), c' \notin Q\}.$$

Then, we compute Q_2 iteratively using the next pseudocode:

$Q_2 = Q' \cup (Ascendants(c_d) \cap Descendants(v_a)) \cap C$
while $Incomplete(Q_2) \neq \emptyset$ **do**
  $Q_2 = Q_2 \cup InitialClass(Incomplete(Q_2))$
end while

We define a *test sequence* as a sequence of test cases $TS = (Q_i)$ with $1 \leqslant i \leqslant n$, where the first test case is the initial one, that is, $Q_1 = Q_{ini}$. Given a test sequence, the class or transition coverage of the sequence is defined as the ratio between the visited classes (or transitions) and all the classes (or transitions). We formally define the coverage criteria used in this paper as follows:

$$ClassCoverage(sol) = \frac{\left|\bigcup_{i=1}^{n} Q_i\right|}{|C|} \tag{A.3}$$

$$TransitionCoverage(sol) = \frac{\left|\bigcup_{i=1}^{n-1} Transitions(Q_i, Q_{i+1})\right|}{|T|} \tag{A.4}$$

where *Transitions* is defined as:

$$Transitions(Q_i, Q_{i+1}) = (Q_i \times Q_{i+1}) \cap T$$

Given an ECTM model, the objective of the TSGP is the generation of a set of test sequences that maximizes any of the coverage criterion (one each time) defined above (class or transition).

Appendix B. Citizen watch model

Fig. B.7.

References

[1] The Economic Impacts of Inadequate Infrastructure for Software Testing, Technical Report; NIST, 2002.
[2] E. Alba, F. Chicano, Finding safety errors with ACO, in: GECCO'07, ACM Press, London, UK, 2007, pp. 1066–1073.
[3] E. Alba, F. Chicano, M. Ferreira, J. Gomez-Pulido, Finding deadlocks in large concurrent java programs using genetic algorithms, in: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, GECCO '08, ACM, New York, NY, USA, 2008, pp. 1735–1742.
[4] A. Arcuri, G. Fraser, Parameter tuning or default values? An empirical investigation in search-based software engineering, Emp. Softw. Eng. 18 (3) (2013) 594–623.
[5] C. Blum, A. Roli, Metaheuristics in combinatorial optimization: overview and conceptual comparison, ACM Comput. Surv. 35 (3) (2003) 268–308.
[6] S. Burton, J. Clark, J. McDermid, Automated generation of tests from statechart specifications, in: FATES'01, 2001, pp. 31–46.
[7] M. Cohen, J. Snyder, G. Rothermel, Testing across configurations: implications for combinatorial testing, SIGSOFT Softw. Eng. Notes 31 (2006) 1–9.
[8] M. Conrad, Systematic testing of embedded automotive software – the classification-tree method for embedded systems, in: Perspectives of Model-

Based Testing, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005, pp. 1–12.
[9] M. Dorigo, T. Stützle, Ant Colony Optimization, The MIT Press, 2004.
[10] E. Dustin, Effective Software Testing: 50 Ways to Improve Your Software Testing, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
[11] A. Gargantini, E. Riccobene, Asm-based testing: coverage criteria and automatic test sequence generation, J. Univ. Comp. Sci. 7 (2001) 1050–1067.
[12] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, Y. Wolfsthal, Coverage-directed test generation using symbolic techniques, in: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design, London, UK, 1996, pp. 143–158.
[13] M. Grochtmann, K. Grimm, Classification trees for partition testing, Softw. Test., Verif. Reliab. 3 (2) (1993) 63–82.
[14] W.J. Gutjahr, K. Doerner, Extracting test sequences from a markov software usage model by ACO, in: E.C.P., et al. (Ed.), Proceedings of GECCO 2003, LNCS, vol. 2724, Springer, 2003, pp. 2465–2476.
[15] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comp. Program. 8 (1987) 231–274.
[16] M. Harman, The current state and future of search based software engineering, in: ICSE/FOSE '07, IEEE Computer Society, Minneapolis, Minnesota, USA, 2007, pp. 342–357.
[17] M.P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, J. Gao, Auto-generating test sequences using model checkers: a case study, in: Proceedings of Formal Approaches to Testing of Software (FATES 2003), 2003, pp. 42–59.
[18] S. Helke, Verifikation von Statecharts durch struktur- und eigenschaftserhaltende Datenabstraktion, Ph.D. thesis; Technische Universität Berlin, 2007.
[19] R.M. Hierons, M. Harman, H. Singh, Automatically generating information from a Z specification to support the Classification Tree Method, in: 3rd International Conference of B and Z Users, LNCS, vol. 2651, 2003, pp. 388–407.
[20] IBM, IBM Rhapsody Examples, 2013.
[21] C. Kaner, C. Cem, K. All, The Impossibility of Complete Testing, 1997.
[22] P.M. Kruse, J. Wegener, Test sequence generation from classification trees, in: IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), 2012, IEEE, 2012, pp. 539–548.
[23] D.R. Kuhn, R.N. Kacker, Y. Lei, Practical Combinatorial Testing, Technical Report, National Institute for Standards and Technology (NIST), 2010.
[24] E. Lehmann, J. Wegener, Test case design by means of the CTE XL, in: EuroSTAR 2000, Kopenhagen, Denmark, 2000, pp. 1–10.
[25] P.J. Lucas, An Object-Oriented Language System For Implementing Concurrent, Hierarchical, Finite State Machines, Ph.D. thesis; Graduate College of the University of Illinois at Urbana–Champaign, 1989.
[26] T. MathWorks, Matlab Simulink Stateflow, 2008.
[27] P. McMinn, Search-based software test data generation: a survey, Softw. Test., Verif. Reliab. 14 (2) (2004) 105–156.
[28] U. Mirosamek, Two Orthogonal Regions (Main Keypad and Numeric Keypad) of a Computer Keyboard, 2009.
[29] A.S. Namin, J.H. Andrews, The influence of size and coverage on test suite effectiveness, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09, ACM, New York, NY, USA, 2009, pp. 57–68.
[30] J. Oh, M. Harman, S. Yoo, Transition coverage testing for Simulink/Stateflow models using messy genetic algorithms, in: GECCO'11, 2011, pp. 1851–1858.
[31] T.J. Ostrand, M.J. Balcer, The category-partition method for specifying and generating fuctional tests, Commun. ACM 31 (1988) 676–686.
[32] D.J. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures, forth ed., Chapman & Hall/CRC, 2007.
[33] J. Torres-Jimenez, E. Rodriguez-Tello, New bounds for binary covering arrays using simulated annealing, Inform. Sci. 185 (2012) 137–152.
[34] H. Ural, Formal methods for test sequence generation, Comput. Commun. 15 (1992) 311–325.
[35] A. Vargha, H.D. Delaney, A critique and improvement of the cl common language effect size statistics of mcgraw and wong, J. Edu. Behav. Statist. 25 (2) (2000) 101–132.
[36] T. Vos, F. Lindlar, B. Wilmes, A. Windisch, A. Baars, P.M. Kruse, H. Gross, J. Wegener, Evolutionary functional black-box testing in an industrial setting, Softw. Qual. Changes (2012).
[37] A.W. Williams, R.L. Probert, A measure for component interaction test coverage, in: Proceedings of the ACS–IEEE International Conference on Computer Systems and Applications, vol. 30, 2001, pp. 301–311.
[38] A. Windisch, Search-based testing of complex simulink models containing stateflow diagrams, in: SBST'08, Lillehammer, Norway, 2008, pp. 251–251.