DOI 10.1007/s00165-011-0206-3
BCS © 2012
Formal Aspects of Computing (2013) 25: 743–768

Formal Aspects of Computing

Testing restorable systems: formal definition and heuristic solution based on river formation dynamics

Pablo Rabanal, Ismael Rodríguez and Fernando Rubio

Dept. Sistemas Informáticos y Computación, Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain.
E-mail: fernando@sip.ucm.es

Abstract. Given a finite state machine denoting the *specification* of a system, finding some short interaction sequences capable of reaching some/all states or transitions of this machine is a typical goal in testing methods. If these sequences are applied to an *implementation under test*, then equivalent states or transitions would be reached and observed in the implementation—provided that the implementation were actually defined as the specification. We study the problem of finding such sequences in the case where configurations previously traversed can be *saved* and *restored* (at some cost). In general, this feature enables sequences to reach the required parts of the machine in less time, because some repetitions can be avoided. However, we show that finding optimal sequences in this case is an NP-hard problem. We propose an heuristic method to approximately solve this problem based on an evolutionary computation approach, in particular *river formation dynamics* (RFD). Given *finite state machine* specifications and sets of states/transitions to be reached, we apply RFD to construct testing plans reaching these configurations. Experimental results show that being able to load previously traversed states generally reduces the time needed to cover the target configurations.

Keywords: Testing, Weighted Finite State Machines, Minimum Load Sequence, Evolutionary Computation, River Formation Dynamics

1. Introduction

Testing developed systems is an industrial necessity in any IT developing process. In particular, *formal testing techniques* [LY96, Pet01, BT01, RMN08, Rod09] allow testers to perform testing tasks in a systematic and (semi-) automatic way. Thus, they reduce the dependency on the inspiration of a concrete human tester. Let us remark that, in general, precisely assessing the correctness of an *implementation under test* (IUT) requires testing an infinite number of available behaviors. If the tester can assume certain *hypotheses* about the IUT (see, e.g. [Hie02, RMN08, Hie09]), or she can assume that faults must belong to a given *fault model* (see, e.g. [PYB96, Pet01]), then the set of systems that could actually *be* the IUT is reduced, so finding out whether the IUT is correct or not (i.e. whether it belongs to the set of possible right systems or to the set of possible wrong systems) is easier. It

Correspondence and offprint requests to: F. Rubio, E-mail: fernando@sip.ucm.es
This paper constitutes and extended and revised version of [RRR09b]. Research partially supported by projects TIN2009-14312-C02-01 and Santander GR35/10-A—group number 910606.

has been observed that the assumption of some hypotheses may enable the existence of finite *complete* test suites, that is, test suites such that observed results allow them to precisely determine whether the IUT is correct or not [BGM91, Gau95]. In the framework presented in [RMN08], the tester can decide which hypotheses, selected from a repertory of testing hypotheses typically appearing in the context of *finite state machines* (FSMs), she actually assumes, and then a *logic* is used to automatically infer whether a given set of observations implies the correctness of the IUT—provided that all assumed hypotheses hold. In [Rod09] the specific conditions that enable the existence of finite complete test suites are studied in a general testing framework. In general, hypotheses and fault models enabling the existence of finite complete test suites are very restrictive and, moreover, finite complete test suites may have exponential size with respect to the size of models. Thus, testers typically refuse to pursue the completeness, that is, they do not aim at applying some tests such that observed results allow to precisely determine whether the IUT is correct or not. Instead, some *coverage testing* criteria are pursued [ZHM97, MA01, HW05], which allows testers to concentrate on testing a part of the IUT behaviors.

For instance, given a specification represented by a FSM, precisely distinguishing it from any other FSM whose number of states is under a given threshold may be a very expensive task. In general, the length of a sequence of inputs allowing to make such a distinction, if it exists, is exponential with the number of states. Techniques such as distinguishing sequence (DS) [Hen64, Gon70, SL89], characterizing set (CS) [Cho78], and unique input/output sequence (UIO) [SD85, SD88] have been proposed in the literature. The generation of test sequences using these techniques are the D-, W-, and U-methods, respectively. In fact, UIOs tend to obtain shorter test sequences than D- and W-methods [WH87, SL88, ENDK02, CJH06, IB07]. Besides, several graph theory techniques have been used to find short test sequences reaching states or transitions in an FSM specification. In particular, the *Chinese Postman* problem (consisting in traversing at least once all edges of a graph) has been used in combination with UIOs or DSs [ADLU88, DU95, SL96].

Still, test sequences distinguishing a machine from any other of a given size are, indeed, much *longer* than other useful test sequences aiming at more modest goals. For instance, let us suppose that, given an FSM specification and an IUT, we compose a test suite that reaches, in such FSM, all transitions or all states. If the IUT behavior were exactly the same as the specification (that is, if its behavior could be defined by an equivalent FSM), then any test suite reaching some states/transitions in the specification would reach equivalent states/transitions in the IUT. However, the IUT could differ from the specification, so reaching a specification configuration by following a given path does not imply that the same configuration is reached in the IUT by following the same path. Therefore, a test suite reaching each specification state or transition by following, for each target state or transition, a single path, is *not* a complete test suite in general. However, such an incomplete test suite could detect a high number of possible faults. Let us assume that the IUT is built by a *nearly competent* implementer, so the probability that the IUT behavior is *similar* to the behavior of the specification is higher than the probability of having a very different behavior. In this case, a high number of test sequences reaching some states/transitions in the specification will reach equivalent states/transitions in the IUT. In particular, the proportion of sequences reaching the required configurations will be higher in general than if *random* input sequences of the same length are applied, because random sequences are not defined on purpose to reach any specific configuration in an IUT being similar to the specification. For instance, if a given area of specification states is reached only by following some long unique path, then random input sequences will likely miss this area and will rather focus on (redundantly) traversing other more accessible areas. On the other hand, input sequences *designed* to reach these states/transitions in the specification will be able to do so in the IUT unless the IUT is wrong in this path (and, if it is so, then tester should be willing to investigate these paths indeed). Thus, if we aim at traversing at least once all (specification) states or transitions, this strategy might broaden the variety of observed (IUT) configurations. Moreover, the size of this kind of test suites is polynomial with the number of states of the FSM. Hence, despite of their incompleteness, these test suites can be very useful in practice.

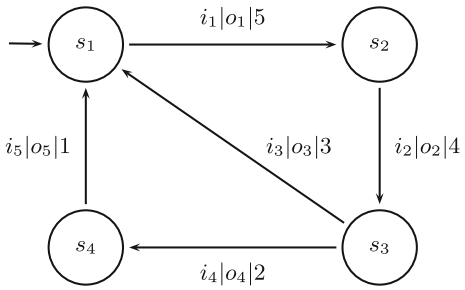

Fig. 1. Example of system

In this paper we generalize testing methods reaching some/all FSM states or transitions to the case where the tester can *restore* any previous configuration of the system. Let us assume that the IUT is a software system and the tester can *save* the complete current configuration of the system at any time. Then, at any subsequent time, the tester could *restore* such configuration and execute the system from that configuration on. In particular, after restoring the configuration she could follow a different path to the one she followed the previous time. Notice that, if configurations can be saved/restored, then the tester can use this feature to avoid *repeating* some execution sequences. Thus, some time assigned to testing activities could be saved. Let us also note that saving/restoring the complete configuration of a system could be time-expensive. These costs are similar to the costs of executing *process suspending* and *process restoring* operations in any operating system, which in turn are due to copying the program state from/to RAM to/from hard disk, respectively. Thus, a testing technique using these features should take these costs into account. Let us consider the (cost-weighted) FSM depicted in Fig. 1. The initial state is s_1 and transitions are labeled by an input, an output, and the time required to take the transition. Let us suppose transitions $s_3 \xrightarrow{i_3|o_3|3} s_1$ and $s_4 \xrightarrow{i_5|o_5|1} s_1$ are *critical*, i.e. they must be tested. The interaction plan $s_1 \xrightarrow{i_1} s_2 \xrightarrow{i_2} s_3 \xrightarrow{i_3} s_1 \xrightarrow{i_1} s_2 \xrightarrow{i_2} s_3 \xrightarrow{i_4} s_4 \xrightarrow{i_5} s_1$ covers these transitions at cost $5+4+3+5+4+2+1 = 24$ time units. Let us suppose saving/loading a previously traversed configuration costs seven time units. Then, the interaction $s_1 \xrightarrow{i_1} s_2 \xrightarrow{i_2} s_3 \xrightarrow{i_3} s_1 \xrightarrow{load\ s_3} s_3 \xrightarrow{i_4} s_4 \xrightarrow{i_5} s_1$ costs $5+4+3+7+2+1 = 22 < 24$, so loading is preferable in this case. In general, saving/restoring operations should be used only if the cost of repeating a path is bigger than the cost of saving and restoring a configuration. Let us note that this is actually the case of many software systems that are computation-intensive but that do not use much memory (e.g. control systems, embedded systems, etc.). Moreover, it might also be the case of other memory-intensive systems as long as data manipulation operations are relatively expensive in time (we will illustrate this in Sect. 7, where we will present a case study involving a database management system). It is also worth to point out that we are implicitly assuming that we can save the state of the IUT but we cannot directly *set up* states. Let us note that setting up states requires understanding the state representation to a high extent, but this is not the case in black-box testing. Still, operating systems use process suspending and process restoring operations to copy the program state from/to RAM to/from hard disk, respectively. These operations are transparent to the user and are performed without requiring any knowledge about the code of suspended/restored programs. Thus, these operations are suitable in a black-box testing approach. Similarly, in some cases the state of registers in a hardware device can be externally read and written, allowing to copy/restore system states without understanding the internal representation of these states.

In this paper we present a methodology that, given (i) the time cost of saving/restoring a configuration; (ii) an FSM specification explicitly denoting the time cost of performing each transition; and (iii) a set of *critical* system configurations that are required to be observed, it returns a plan to sequentially interact with the system (possibly including saving/restoring operations) that allows all critical configurations to be reached in a low overall time. As we will show, obtaining the *optimal* interaction sequence is an NP-hard problem. Thus, instead of an exact solution, we will develop an heuristic method to approximate the optimal path. Let us note that an interaction plan where saving/restoring operations are allowed (under a certain cost) is, in fact, a plan where each save/restore point represents a *bifurcation*: initially, we will take a path, and then we will return to a previous point and follow another path. We will prove that considering interaction *trees*, instead of interaction sequences, does not restrict the possibility to find good paths. In particular, we will show that, for every path σ, there exists an equivalent path that can be represented by a tree, which implies that all steps of σ can be sorted in such a way that each load refers *only* to states appearing in the *same* branch of some tree. Consequently, interaction plans constructed by

our method will not be represented by a sequential path, but by a tree covering all critical points. That is, our goal will be finding a tree where the sum of the costs of all the transitions of the tree (including the sum of the costs of saving/restoring operations) is minimal.

In order to heuristically solve this NP-hard problem, we use an *evolutionary computation* approach [Jon06], in particular *river formation dynamics* (RFD) [RRR07, RRR09a]. This method has already been used to solve other NP-complete problems consisting in constructing other kinds of *covering trees* [RRR08]. Briefly, RFD consists in simulating how drops transform the landscape while they flow down to the sea. Drops erode the ground when they traverse high decreasing gradients, and they deposit carried sediments in flatter areas. By increasing/decreasing the altitude of nodes, gradients are modified, which in turn affects movements of subsequent drops. Eventually, formed decreasing gradients will depict *paths* from the point(s) where it rains to the sea. Since drop movements are driven by the *difference* of some values (in particular, node altitudes) rather than by these values themselves, RFD can be seen as a *gradient-oriented* version of *ant colony optimization* (ACO) [Dor04]. However, there are other important differences between RFD and ACO (see [RRR08, RRR09a] for details). In particular, gradients implicitly avoid that a drop follows a local cycle (it is impossible that a path from A to A is ever-decreasing), shortcuts are quickly reinforced because they connect the same origin and destination as older paths, but at lower cost (thus, the decreasing gradient is immediately higher and edges in the shortcut are immediately preferable in average), and the sediment process provides a focused way to *punish* wrong paths (if drops get blocked then they leave sediments and *increase* the node altitude, which eventually prevents other drops from following the same path).

The rest of the paper is structured as follows. Some related work in the domain of *search based software engineering* and *search based testing* are discussed in the next section. In Sect. 3 we formally describe the problem to be solved. In Sect. 4 we introduce our general heuristic method, while in Sect. 5 we show how to apply it to our particular problem. Afterwards, in Sect. 6 we report the results obtained in some randomly generated experimental cases. A more realistic case study, where a part of a database management system is tested, is presented in Sect. 7. Finally, in Sect. 8 we present our conclusions and some lines of future work. Proofs of results are given in the Appendix of the paper.

2. Related work

There exists many works on the application of search based optimization techniques in different software engineering (SE) activities involving project planning, cost estimation, testing, compiler optimization, etc. (see, e.g. [Bot02, APH04, BL05, CKSa06, Har07]). One SE area where optimization techniques are used more prolifically is software testing. In this area, the most used techniques are local search (hill climbing [HHP02]), simulated annealing [MRR+53], genetic algorithms [Hol75] and ACO [DG03, SR09]. Classic techniques have received little attention for SE problems. For example, linear programming [Dan63] or branch and bound [LW66] have been used as optimization techniques to calculate the optimal solution. However, these algorithms are often impracticable in real problems. In fact, most of these problems are versions of NP-complete problems. This is the main motivation for using a metaheuristic search.

A concrete area where metaheuristic search has been widely applied is model-checking. In [AC07] Alba and Chicano propose using ACO to refute LTL formulae in concurrent systems. Their results show that ACO algorithms outperform exhaustive methods in efficiency and efficacy. In other papers, classical algorithms have been used to verify safety properties exploring graphs. For example, Edelkamp et al. [ELLL01] applied A*, Weighted A*, Iterative Deepening A*, and Best First Search to this problem using SPIN and they compared the results with the results obtained using a heuristic search. The authors show that the length of the counterexamples is shortened by using the heuristic search and the amount of memory required is reduced. Genetic algorithms (GA) have also been applied for refuting safety properties (see [AT96, GK02]).

One of the fields in SE where optimization techniques have been broadly applied is software testing, in particular to test data generation [Kor92, PHP99, McM04, HM09]. Software testing is very important to detect and reduce system errors especially in critical software. In particular, improving the testing infrastructure can save more than $\frac{1}{3}$ of the costs generated by software failures (see [Tas02]), so testing automation is essential. A typical problem in software testing consists in finding an input to lead the software to a certain state. Search based optimization techniques have been applied to solve the problem of test data generation. This approach is known as search based testing (SBT). SBT has been successfully applied to FSM testing, mutation testing, temporal testing, structural testing, etc. (see, e.g. [LY94, WSJE97, MMS01, Hie04, LHJ09]). In SBT the search is commonly guided by an evolutionary algorithm.

Lots of systems are modeled by using FSMs. A typical task to ensure the reliability of a system consists in testing its conformance to a specification by applying a sequence of inputs and verifying that the outputs are correct. Next we comment some works on SBT applied to FSMs. In [DHHG06] Derderian et al. describe a method to automatically generate unique input output (UIO) sequences for each state of a FSM. They solve this NP-hard problem by using a GA method which can work without a fully specified FSM and obtain good results. In [LY08] Lehre and Yao analyze the influence of genetic operators on the problem of generating the UIO sequences for FSMs. The authors study if the settings of an evolutionary algorithm have an important impact when finding UIOs and prove that crossover can be essential for solving this problem in polynomial time. Lam et al. in [LJL07] present an approach based on the Wp-method [LvBP94, ENDK02] to minimize the length of the test sequences. The method reformulates the problem by using ACO to solve the asymmetric traveling salesman problem. Finally, the authors show that the approach maintains the same fault detection capability than the Wp-method.

3. Problem definition

In this section we formally introduce the problem to be solved as well as some related notions. Next we introduce some preliminary notation:

- Given a pair $x = (a, b)$, we assume $\texttt{fst}(x) = a$ and $\texttt{snd}(x) = b$.
- Given a list $l = [x_1, \ldots, x_n]$, we represent by $l[i]$ the ith element of l, i.e. $l[i] = x_i$. Besides, $\texttt{length}(l)$ returns the number of elements of l, i.e. $\texttt{length}(l) = n$.

We introduce the formalism used to define specifications, called *weighted finite state machine* (WFSM). This is an FSM where the cost of restoring a previous configuration, as well as the cost of taking each transition, are explicitly denoted.

Definition 3.1 A *weighted finite state machine* (from now on WFSM) is a tuple $(S, s_{in}, I, O, C, \Delta)$ where

- S is a finite set of states and $s_{in} \in S$ is the initial state.
- I and O are the finite sets of *input* and *output* actions, respectively.
- $C \in \mathbb{N}$ is the *cost of restoring* a previously traversed state.
- $\Delta \subseteq S \times S \times I \times O \times \mathbb{N}$ is the set of transitions. A transition $\delta \in \Delta$ is a tuple (s_1, s_2, i, o, c) where s_1 is the origin state, s_2 is the destination state, i is the input that triggers the transition, o is the output produced by the transition, and c is a positive natural value that represents the cost of the transition. We write $s_1 \xrightarrow{i/o/c} s_2$ as shorthand for $(s_1, s_2, i, o, c) \in \Delta$.

  A WFSM is *deterministic* if for all $s \in S$ and $i \in I$ we have

  $$\| \{s \xrightarrow{i/o/c} s' \mid \exists\ o, c, s' : s \xrightarrow{i/o/c} s' \in \Delta\} \| \leq 1$$

□

We will assume that specifications are defined by deterministic WFSMs. From now on, in all definitions we will assume that a WFSM $W = (S, s_0, I, O, C, \Delta)$ is implicitly given.

Executions of WFSMs will be denoted by *load sequences*. Each step in a load sequence consists in either taking some WFSMtransition or restoring a previously traversed configuration. The latter action will be denoted by a symbol $\psi(s_k)$ in the sequence meaning that, at the current step of the sequence, we move from the current state to a previously traversed state s_k by *loading* it (instead of by traversing actual transitions of the WFSM). The goal of our method will be finding the cheapest load sequence belonging to the set of all load sequences that covers some given states and/or transitions. In order to reduce the searching space, we will constrain the notion of load sequence in such a way that some *useless* sequences are discarded from scratch. In particular, a condition will be imposed to *ban* some sequences that are equivalent, in terms of cost, to other available sequences that do fulfill the condition. Load sequences (regardless of whether they fulfill the additional condition or not) and load sequences that *actually* fulfill it will be called *load sequences* and α-*load sequences*, respectively. Intuitively, α-sequences are sequences that could be properly represented by some kind of *tree*. Later we will represent some kind of FSM traversals by *trees* where nodes represent states and arcs represent transitions. In these trees, leaves represent states where we load a previously traversed state. The loaded state must be represented by a node

traversed in the same branch as the leaf. After loading that node, a different path will be taken, so this node will be a *branching node* in the tree. In fact, the set of (load-based) FSM traversals represented by a tree will be given by the set of possible pre-order traversals of the tree.[1] Intuitively, α-sequences are sequences that could be a pre-order traversal of some of these trees. Later we will show that constraining our search to considering only α-sequences, rather than considering any kind of load sequences, does not limit the possibility to find cheap sequences.

Definition 3.2 A *load sequence* of W is a sequence $\sigma = s_1 \xrightarrow{\delta_2} s_2 \ldots s_{n-1} \xrightarrow{\delta_n} s_n$ where, for each $2 \leq k \leq n$, δ_k is either $i_k/o_k/c_k$ (if $s_{k-1} \xrightarrow{i_k/o_k/c_k} s_k \in \Delta$) or $\psi(s_j)$ (if $s_j \in \{s_1, \ldots, s_{k-1}\}$ and $s_k = s_j$). In addition, σ is also an *α-load sequence* if for all $s, s' \in S$ such that the first appearance of s in σ is before the first appearance of s' (i.e. such that we have $s = s_i$ and $s' = s_j$ for some $i < j$ such that $s_k \neq s$ for all $k < i$ and $s_l \neq s'$ for all $l < j$) all loads to s appear either after the last load to s' or before s' appears (i.e. there do not exist δ_p and δ_q with $j < p < q$ such that $\delta_p = \psi(s)$ and $\delta_q = \psi(s')$).

The set of all load sequences of W is denoted by $\texttt{Sequences}(W)$. The set of all α-load sequences of W is denoted by $\alpha\texttt{-Sequences}(W)$.

Given $\sigma = s_1 \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_{n-2}} s_{n-1} \xrightarrow{\delta_{n-1}} s_n \in \texttt{Sequences}(W)$, we consider $\sigma^- = s_1 \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_{n-2}} s_{n-1}$. Besides, we consider ${}^-\sigma = s_2 \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_{n-1}} s_n$.

Let $\sigma_1, \cdots, \sigma_n \in \texttt{Sequences}(W)$ be such that for all $1 \leq i \leq n$ we have $\sigma_i = s_{i,1} \xrightarrow{\delta_{i,1}} \cdots \xrightarrow{\delta_{i,k_i-1}} s_{i,k_i}$ and for all $1 \leq i \leq n-1$ we have $s_{i,k_i} = s_{i+1,1}$. The *concatenation* of $\sigma_1, \cdots, \sigma_n$, denoted by $\sigma_1 \cdot \cdots \cdot \sigma_n$, is defined as $s_{1,1} \xrightarrow{\delta_{1,1}} \cdots \xrightarrow{\delta_{1,k_1-1}} s_{1,k_1} \xrightarrow{\delta_{2,1}} s_{2,1} \xrightarrow{\delta_{2,2}} \cdots \xrightarrow{\delta_{2,k_2-1}} s_{2,k_2} \cdots \xrightarrow{\delta_{n,k_n-1}} s_{n,k_n}$. □

Clearly, $\alpha\texttt{-Sequences}(W) \subseteq \texttt{Sequences}(W)$. For instance, let us consider the following sequence $\sigma_1 = s_1 \xrightarrow{i_1/o_1/c_1} s_2 \xrightarrow{i_2/o_2/c_2} s_3 \xrightarrow{\psi(s_1)} s_1 \xrightarrow{i_3/o_3/c_3} s_4 \xrightarrow{\psi(s_2)} s_2 \xrightarrow{i_4/o_4/c_4} s_5$. We have $\sigma_1 \in \texttt{Sequences}(W)$ for some WFSM W, but $\sigma_1 \notin \alpha\texttt{-Sequences}(W)$ (see states s_1 and s_2 and the positions of $\psi(s_1)$ and $\psi(s_2)$ in σ_1). Let us consider $\sigma_2 = s_1 \xrightarrow{i_5/o_5/c_5} s_6 \xrightarrow{i_7/o_7/c_7} s_7 \xrightarrow{i_8/o_8/c_8} s_8 \xrightarrow{\psi(s_7)} s_7 \xrightarrow{i_9/o_9/c_9} s_{10} \xrightarrow{\psi(s_6)} s_6$. We have $\sigma_2 \in \alpha\texttt{-Sequences}(W)$ (provided that these transitions also exist in W).

The *cost* of a load sequence is given by the addition of transition costs and load costs. Besides, the state (respectively, transition) *coverage* of a load sequence is the set of states (resp. transitions) appearing in the sequence. Note that a state (or transition) can appear several times in a given load sequence. There are two possible reasons for this: either a previous state is *loaded*, or an already traversed state/transition is reached again by traversing some transitions. The latter possibility is useful if we want to come back to some state and we realize that doing it *manually* (i.e. by taking transitions) is cheaper than making a *load*. Note that, since state and transition coverages are denoted by *sets* (not multisets), each state or transition appears at most once in the corresponding coverage set.

Definition 3.3 Let $\sigma = s_1 \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_{n-1}} s_n \in \texttt{Sequences}(W)$. If $1 \leq k \leq n-1$ then the *cost* of δ_k, denoted by $\texttt{CostTran}(\delta_k)$, is defined as

$$\texttt{CostTran}(\delta_k) = \begin{cases} c_k & \text{if} \quad \delta_k = i_k/o_k/c_k \\ C & \text{if} \quad \delta_k = \psi(s_j) \end{cases}$$

The *cost* of σ, denoted by $\texttt{CostSeq}(\sigma)$, is defined as $\sum_{k=1}^{n-1} \texttt{CostTran}(\delta_k)$. The *states coverage* or σ, denoted by $\texttt{StateCover}(\sigma)$, is defined as $\texttt{StateCover}(\sigma) = \{s_i \mid 1 \leq i \leq n\}$. The *transitions coverage* of σ, denoted by $\texttt{TranCover}(\sigma)$, is defined by $\texttt{TranCover}(\sigma) = \{s_j \xrightarrow{i_j/o_j/c_j} s_{j+1} \mid 1 \leq j \leq n-1 \wedge \delta_j = i_j/o_j/c_j\}$. □

Given the load sequence σ_1 considered before, $\texttt{CostSeq}(\sigma_1) = c_1 + c_2 + C + c_3 + C + c_4$. Besides, $\texttt{StateCover}(\sigma_1) = \{s_1, s_2, s_3, s_4, s_5\}$ and $\texttt{TranCover}(\sigma_1) = \{s_1 \xrightarrow{i_1/o_1/c_1} s_2, s_2 \xrightarrow{i_2/o_2/c_2} s_3, s_1 \xrightarrow{i_3/o_3/c_3} s_4, s_2 \xrightarrow{i_4/o_4/c_4} s_5\}$. Next we define the target problem of this paper.

[1] There might exist several pre-order traversals because no order between branches (e.g. from left to right) is assumed.

Definition 3.4 Given a WFSM $W = (S, s_0, I, O, C, \Delta)$, some sets $S' \subseteq S$ and $\Delta' \subseteq \Delta$, and a natural number $K \in \mathbb{N}$, the *minimum load sequence* problem, denoted by MLS, is stated as follows: is there a load sequence $\sigma \in \texttt{Sequences}(W)$ starting by s_0 such that $\texttt{CostSeq}(\sigma) \leq K$, $S' \subseteq \texttt{StateCover}(\sigma)$, and $\Delta' \subseteq \texttt{TranCover}(\sigma)$?

To the best of our knowledge, MLS has not been defined or studied before in the literature. Thus, before describing our method to solve it, we prove that it is an NP-complete problem. The proofs of all results presented in this paper are given in the Appendix.

Theorem 3.1 $\texttt{MLS} \in \texttt{NP-complete}$.

Next we show that by constraining our search for good testing sequences to α-load sequences (instead of considering all load sequences) we do not lose the possibility to find sequences whose cost is under some given upper bound.

Proposition 3.1 *For all* $\sigma \in \texttt{Sequences}(W)$ *there exists* $\sigma' \in \alpha\texttt{-Sequences}(W)$ *such that* $\texttt{CostSeq}(\sigma) = \texttt{CostSeq}(\sigma')$, $\texttt{StateCover}(\sigma) = \texttt{StateCover}(\sigma')$, *and* $\texttt{TranCover}(\sigma) = \texttt{TranCover}(\sigma')$. □

Let us revisit σ_1. Since $\sigma_1 \in \texttt{Sequences}(W)$, we have that $\sigma_3 \in \alpha\texttt{-Sequences}(W)$ for sequence $\sigma_3 = s_1 \xrightarrow{i_1/o_1/c_1} s_2 \xrightarrow{i_2/o_2/c_2} s_3 \xrightarrow{\psi(s_2)} s_2 \xrightarrow{i_4/o_4/c_4} s_5 \xrightarrow{\psi(s_1)} s_1 \xrightarrow{i_3/o_3/c_3} s_4$. The properties $\texttt{CostSeq}(\sigma_1) = \texttt{CostSeq}(\sigma_3)$, $\texttt{StateCover}(\sigma_1) = \texttt{StateCover}(\sigma_3)$, and $\texttt{TranCover}(\sigma_1) = \texttt{TranCover}(\sigma_3)$ trivially hold because we traverse the same transitions and we make the same number of loads.

The condition imposed to load sequences to be considered as α-load sequences implies, in particular, that α-load sequences can be equivalently represented as *trees*. A *load tree* is a tree-like version of an α-load sequence, where loads are represented by *bifurcations* in the tree. In particular, any node with several children represents that, after we complete one of the choices represented by these children (i.e. after we complete the *subtree* representing this choice), we will restore the state represented by that node and we will go on through another choice (another subtree). As in the case of sequences, states and transitions can appear several times in the tree. Since loads are represented *only* by bifurcations, a repeated appearance of a state represents that this state is reached again by taking some transitions, rather than by loading. Let us note that, according to the next definition, a leaf can be represented by a tree $t = (node, [\,])$, i.e. a node without children.

Definition 3.5 A *load tree* of W is a term t belonging to the language induced by the term Lt in the following E-BNF:

$$
\begin{array}{lll}
Lt ::= & (St, Ch) & \\
St ::= & s_1 \mid \ldots \mid s_n & \text{where } S = \{s_1, \ldots, s_n\} \\
Ch ::= & [(Tr, Lt), \ldots, (Tr, Lt)] & \\
Tr := & (i, o, n) & \text{where } (i, o, n) \in I \times O \times \mathbb{N}
\end{array}
$$

such that, if t follows the form $t = (st, [(tr_1, child_1), \ldots, (tr_n, child_n)])$, then for all $1 \leq k \leq n$ we have $st \xrightarrow{tr_k} \texttt{fst}(child_k) \in \Delta$.

The set of all load trees for a WFSM W is denoted by $\texttt{Trees}(W)$. □

Next we formally define the *cost* of a load tree $t = (root, children)$. In the next definition, the term $C \cdot (n-1)$ represents the cost spent in restoring the state $root$ in t: assuming t has n children, $n-1$ loads are necessary to come back from the last state visited by each child to $root$. In the next recursive definition, the anchor case is reached when the list of children is empty, i.e. when we have $t = (root, [\,])$ (note that, in this case, no recursive calls are made).

Definition 3.6 The cost of a tree $t = (root, children)$, denoted by $\texttt{CostTree}(t)$, is defined as

$$
\texttt{CostTree}(t) = \sum_{k=1}^{n} \left(\begin{array}{l} \texttt{CostTran}(\texttt{fst}(children[k])) + \\ \texttt{CostTree}(\texttt{snd}(children[k])) \end{array} \right) + C \cdot (n-1)
$$

where $n = \texttt{length}(children)$. □

We define a boolean predicate $\sigma \rhd t$ returning *true* if the sequence σ *corresponds* to the tree t, that is, if σ could be a sequential realization of the plan described by t. Intuitively, the sequence must be one of the possible *pre-order* traversals of the tree. More technically, in order to compare the sequence σ and the tree t, we transform σ into a tree and next we compare the resulting tree with t. In order to create a tree from a sequence σ, we set the first state s_1 of the sequence as the root of the tree. Then, we split the rest of the sequence into *subsequences*, dividing them at the places where we load s_1. Next we recursively construct the tree of each of these subsequences, and the returned trees are taken as children of t. In functions `createTree` and `createSeq` given in the next definition, anchor cases are again reached when the target tree has no children. Besides, note that, in function `createTree`, the terms $^-(\sigma_1^-), \ldots, ^-(\sigma_n^-)$ denote that the first and the last steps of sequences $\sigma_1, \ldots, \sigma_n$ are removed before the recursive call is done. Similarly, in function `createSeq`, the term σ^- denotes that the last step of the sequence is removed. This step is the last load to *root*, so this load in unnecessary.

Definition 3.7 Let $t_1 = (root_1, children_1)$ and $t_2 = (root_2, children_2)$ with $t_1, t_2 \in \texttt{Trees}(W)$. Let $n_1 = \texttt{length}(children_1)$ and $n_2 = \texttt{length}(children_2)$. We say that t_1 and t_2 are *equivalent*, denoted by $t_1 \equiv_T t_2$, if

(1) $root_1 = root_2$,
(2) $n_1 = n_2$,
(3) There exists a bijection $f : [1..n_1] \longrightarrow [1..n_2]$ such that, for all $1 \leq i \leq n_1$, we have $\texttt{fst}(children_1[i]) = \texttt{fst}(children_2[f(i)])$ (i.e. transitions leading to each children coincide) and we have $\texttt{snd}(children_1[i]) \equiv_T \texttt{snd}(children_2[f(i)])$ (i.e. subtrees are equivalent).

Let $\sigma = \sigma_1 \cdot \cdots \cdot \sigma_n \in \alpha\text{-}\texttt{Sequences}(W)$ be such that for all $1 \leq i \leq n$ we have $\sigma_i = s_1 \xrightarrow{\delta_{1,i}} \cdots \xrightarrow{\delta_{k_i,i}} s_1$, where $\delta_{k_i,i} = \psi(s_1)$ and for all $1 \leq j < k_i$ we have $\delta_{j,i} \neq \psi(s_1)$. The *tree of* σ, denoted by $\texttt{createTree}(\sigma)$, is defined as

$$\texttt{createTree}(\sigma) = (s_1, [(\delta_{1,1}, \texttt{createTree}(^-(\sigma_1^-))), \ldots, (\delta_{1,n}, \texttt{createTree}(^-(\sigma_n^-)))])$$

Let $\sigma \in \alpha\text{-}\texttt{Sequences}(W)$ and $t \in \texttt{Trees}(W)$. We say that σ *corresponds to* t, denoted by $\sigma \rhd t$, if $\texttt{createTree}(\sigma) \equiv_T t$.

Given a tree $t = (root, children)$, the *set of* α*-load sequences* of t, denoted by $\texttt{createSeq}(t)$, is defined as $\{root\}$ if $children = [\,]$; otherwise

$$\texttt{createSeq}(t) = \left\{ \sigma^- \;\middle|\; \begin{array}{l} \sigma = \sigma_{f(1)} \cdot \cdots \cdot \sigma_{f(n)} \;\wedge\; n = \texttt{length}(children) \;\wedge \\ f : [1..n] \longrightarrow [1..n] \text{ is a bijective function} \;\wedge \\ \forall\; 1 \leq i \leq n : \\ \left(\begin{array}{l} \sigma_i = root \xrightarrow{\texttt{fst}(children[i])} \sigma_i' \xrightarrow{\psi(root)} root \;\wedge \\ \sigma_i' \in \texttt{createSeq}(\texttt{snd}(children[i])) \end{array} \right) \end{array} \right\}$$

We define the *state coverage* of a *tree* $t = (root, [(\delta_1, t_1), \ldots, (\delta_n, t_n)])$ as follows: $\texttt{StateCoverT}(t) = \{root\} \cup \bigcup_{i=1}^{n} \texttt{StateCoverT}(t_i)$. We define the *transitions coverage* of t as: $\texttt{TranCoverT}(t) = \{\delta_1, \ldots, \delta_n\} \cup \bigcup_{i=1}^{n} \texttt{TranCoverT}(t_i)$.

Proposition 3.2 *We have the following properties:*

(a) $\texttt{createSeq}(t) = \{\sigma \mid \sigma \rhd t\}$
(b) *for all* $\sigma, \sigma' \in \texttt{createSeq}(t)$ *we have* $\texttt{CostSeq}(\sigma) = \texttt{CostSeq}(\sigma') = \texttt{CostTree}(t)$, $\texttt{StateCover}(\sigma) = \texttt{StateCover}(\sigma') = \texttt{StateCoverT}(t)$, *and* $\texttt{TranCover}(\sigma) = \texttt{TranCover}(\sigma') = \texttt{TranCoverT}(t)$.

□

The previous result guarantees that, if we want to find the cheapest α-load sequence for a given `WFSM`, then we can concentrate on searching for the equivalent cheapest load tree. Given such cheapest tree t, the α-load sequence to be used can be any sequence σ such that $\sigma \in \texttt{createSeq}(t)$. Recall that Proposition 3.1 showed that, if we want to search for a good load sequence, we can concentrate on considering α-load sequences. We conclude by transitivity that, if we want to search for a good load sequence, we can focus on searching a good *load tree*. This idea will be exploited later in Sect. 5, where we will apply an evolutionary computation approach to find trees rather than sequences.

4. Brief introduction to river formation dynamics

In this section we briefly introduce the basic structure of RFD (for further details,see [RRR07, RRR09a]). Given a working graph, we associate *altitude* values to nodes. *Drops* erode the ground (they reduce the altitude of nodes) or deposit the sediment (increase it) as they move. The probability of the drop to take a given edge instead of others is proportional to the gradient of the down slope in the edge, which in turn depends on the difference of altitudes between both nodes and the distance (i.e. the *cost* of the edge). At the beginning, a flat environment is provided, that is, all nodes have the same altitude. The exception is the destination node, which is a hole (the *sea*). Drops are unleashed (i.e. it *rains*) at the origin node(s), and they spread around the flat environment until some of them fall in the destination node. This erodes adjacent nodes, which creates new down slopes, and in this way the erosion process is propagated. New drops are inserted in the origin node(s) to transform paths and reinforce the erosion of promising paths. After some steps, good paths from the origin(s) to the destination are found. These paths are given in the form of sequences of decreasing edges from the origin to the destination. Several improvements are applied to this basic general scheme (see [RRR07, RRR09a]).

Next we present in detail the basic scheme of the RFD algorithm:

```
initializeNodes()
initializeDrops()
while (not allDropsFollowTheSamePath()) and (not otherEndingCondition())
   moveDrops()
   erodePaths()
   depositSediments()
   analyzePaths()
end while
```

We comment on the behavior of each step. Given a working graph, we associate *altitude* values to nodes in the `initializeNodes()` phase. At the beginning, a flat environment is provided, that is, all nodes have the same altitude. The exception is the destination node, whose altitude is set to 0 (it represents the *sea*). Then, drops are initialized (`initializeDrops()`), i.e. drops are unleashed (i.e. it *rains*) at the origin node(s).

The `while` loop of the algorithm is executed until either all drops find the same solution (`allDropsFollowTheSamePath()`), that is, all drops departing from the same initial nodes traverse the same sequences of nodes, or another alternative finishing condition is satisfied (`otherEndingCondition()`). This condition may be used, for example, for limiting the number of iterations or the execution time. Another choice is to finish the loop if the best solution found so far is not surpassed during the last n iterations.

In the first step of the loop body (`moveDrops()`) drops spread around the flat environment until some of them fall in the destination node. Drops are moved across the nodes of the graph in a partially random way. The probability of the drop to take a given edge instead of others is proportional to the gradient of the down slope in the edge, which in turn depends on the difference of altitudes between both nodes and the distance (i.e. the *cost* of the edge). The following *transition rule* defines the probability that a drop k at a node i chooses the node j to move next:

$$P_k(i,j) = \begin{cases} \frac{decreasingGradient(i,j)}{\sum_{l \in V_k(i)} decreasingGradient(i,l)} & \text{if} \quad j \in V_k(i) \\ 0 & \text{if} \quad j \notin V_k(i) \end{cases} \tag{1}$$

where $V_k(i)$ is the set of nodes that are *neighbors* of node i that can be visited by the drop k and have a negative value of *decreasingGradient(i,j)*, which represents the gradient between nodes i and j and is defined as follows:

$$decreasingGradient(i,j) = \frac{altitude(j) - altitude(i)}{distance(i,j)} \tag{2}$$

where *altitude(x)* is the altitude of the node x and *distance(i,j)* is the length of the edge connecting node i and node j.

In the next step (`erodePaths()`), *drops* erode adjacent nodes (they reduce the altitude of nodes) depending on the gradient between the origin and the destination node which creates new down slopes, and in this way the erosion process is propagated. Once the erosion process finishes, the altitude of all nodes of the graph is slightly increased (`depositSediments()`). In particular, the altitude of a node is increased according to the sediments eroded in the previous phase.

Finally, the last step (`analyzePaths()`) studies all solutions found by drops and stores the best solution found so far. After that, new drops are inserted in the origin node(s) to transform paths and reinforce the erosion of promising paths. After some steps, good paths from the origin(s) to the destination are found. These paths are given in the form of sequences of decreasing edges from the origin to the destination.

Compared to a related well-known evolutionary computation method, *ant colony optimization* (ACO), RFD provides some advantages that were briefly outlined in the introduction. On the one hand, local cycles are not created and reinforced because they would imply an *ever decreasing cycle*, which is contradictory. Though ants usually take into account their past path to avoid repeating nodes, they cannot avoid being led by pheromone trails through some edges in such a way that a node must be repeated in the next step.[2] However, *altitudes* cannot lead drops to these situations. Moreover, since drops do not have to worry about following cycles, in general drops do not need to be endowed with *memory* of previous movements, which releases some computational memory and reduces some execution time. On the other hand, when a shorter path is found in RFD, the subsequent reinforcement of the path is fast: since the same origin and destination are concerned in both the old and the new path, the difference of altitude is the same but the distance is different. Hence, the edges of the shorter path necessarily have higher down slopes and are immediately preferred (on average) by subsequent drops. Finally, the erosion process provides a method to avoid inefficient solutions because sediments tend to be cumulated in blind alleys (in our case, in *valleys*). These nodes are filled until eventually their altitude matches adjacent nodes, i.e. the valley disappears. This differs from typical methods to reduce pheromone trails in ACO: usually, the trails of *all* edges are periodically reduced at the same rate. On the contrary, RFD intrinsically provides a *focused* punishment of bad paths where, in particular, those nodes blocking alternative paths are modified.

When there are several departing points (i.e. it rains at several points), RFD does not tend in general to provide the shortest path (i.e. river) from each point to the sea. Instead, as happens in nature, it tends to provide a tradeoff between quickly gathering individual paths into a small number of main flows (which minimizes the total size of the formed *tree* of tributaries) and actually forming short paths from each point to the sea. For instance, meanders are caused by the former goal: we deviate from the shortest path just to collect drops from a different area, thus reducing the number of flows. On the other hand, new tributaries are caused by the latter one: by *not* joining the main flows, we can form tailored short paths.[3] These characteristics make RFD a good heuristic method to solve problems consisting in forming a kind of covering tree [RRR08], which motivates using RFD to solve `MLS`.

In [RRR07, RRR09a], several improvements are applied to the basic RFD general scheme. In particular, drops coming from different origins and coinciding at the same node are joined into a bigger drop, which allows to reduce the number of individual drop movements. Besides, in order to avoid the formation of local optima, drops are given a small probability to climb *increasing* slopes. This probability is inversely proportional to the increasing gradient, and it is reduced during the execution of the algorithm. This feature improves the search for good paths because it partially decouples the algorithm from the tendencies imposed at the very first steps (see [RRR07, RRR09a] for further details).

5. Applying RFD to solve `MLS`

In this section we show how the general RFD scheme, described in the previous section, is adapted to solve `MLS`. First, let us note that our goal will be constructing load trees where the root is the *initial state* of the given `WFSM`. In terms of RFD, we will use this node as the final goal of all drops, i.e. the *sea*. In order to make drops go in this direction, each transition of the `WFSM` will be represented in the working graph of RFD by an edge leading to the *opposite* direction. Thus, final trees constructed by RFD will have to be inverted in order to constitute valid `MLS` solutions. Besides, since solutions consist in paths covering some configurations, a modification must be introduced to avoid drops *skip* some required steps. Let us suppose that we have to cover nodes A, B, and C and the optimal path is $A \rightarrow B \rightarrow C$. If this path were formed by RFD then the altitudes x_A, x_B, x_C of these nodes would be such that $x_A > x_B > x_C$. Let us suppose there also exists an edge $A \rightarrow C$. Then, drops will tend to prefer going directly from A to C, which is not optimal because then B is not covered. To avoid this problem, a node will be inserted at each *edge*. In particular, going from A to C will actually imply going from A to a new node ac and next going from ac to C. Since this choice provides an *incomplete* covering, drops following this path

[2] Usually, this implies either to repeat a node or to *kill* the ant. In both cases, the last movements of the ant were useless.

[3] We can make RFD tend towards either of these choices by changing a single parameter (see [RRR08]).

will not be successful and the erosion in this path will be low. Thus, the altitude of ac will remain high and hence taking $A \rightarrow B \rightarrow C$ (in fact, $A \rightarrow ab \rightarrow B \rightarrow bc \rightarrow C$) will be preferable for drops. These additional nodes will be called *barrier nodes*. In terms of MLS, barrier nodes will represent WFSM *transitions*, while standard nodes will represent WFSM *states*. During the execution of RFD, new drops will be introduced in nodes representing critical states and transitions (i.e. it *rains* in these nodes). After executing RFD for some time, a solution tree will be constructed by taking, for each critical node, the path leading to the sea through the highest available decreasing gradient. This policy guarantees that the subgraph depicted by these paths is a tree indeed; if it were not a tree then, for some node N, two outgoing edges would be included in the subgraph. This is not possible because only the edge having the highest gradient is taken.

Let us note that load trees may include repeated states. In particular, a repeated state denotes that we return to a previously traversed state by taking transitions instead of by loading (a load is simply represented by a bifurcation). Thus, solutions constructed by RFD must be able to include repetitions as well. If repetitions are represented by actually making drops pass more than once through the same node in the working graph (and, perhaps, leaving it through a different edge each time), then formed solutions will not be *stable*: in the long term, only one of the edges leaving each node would be *reinforced* by drops, and thus only this edge would prevail. This argument applies to similar methods, such as ACO: in the long term, only one choice would be reinforced by ants. In RFD, there is an additional reason for needing an alternative way to denote repetitions: the formation of gradients implicitly makes RFD avoid following a path from a node to itself. Thus, the general RFD scheme must be adapted. In particular, the working graph of RFD will be modified. One possibility consists in introducing several instances of each state in the graph, so that each instance can have its own altitude. In this case, paths formed by RFD could go from an instance of a node to another instance of the *same* node, and these paths would explicitly denote not only state repetitions, but also *when* states must be repeated. Let us note that nodes may be repeated more than twice in a load tree, so this solution would force us to significantly increase the number of nodes of the working graph. Instead, an alternative solution will be applied. Let us note that the purpose of repeating some nodes through transitions (i.e. not by using loads) is reaching some state or transition that has not been traversed before. In particular, we will never load *right after* traversing some repeated states and transitions: directly loading, instead of traversing some repeated states and transitions and next loading, would have the same effect in terms of covering target entities, but at a lower cost (recall that the load cost C does not depend on the current state or the state to be loaded). Thus, our target tree does not *need* to repeat states or transitions; it just needs to be able to reach the destinations we could reach if states or transitions could be repeated.

In order to allow this, *additional edges* connecting each state with the rest of the states through the *shortest available path* will be added to the graph. Let us suppose that we wish to go from A to an (unvisited) critical node B, and the cheapest way to do it is traversing some (repeated and/or non-critical) nodes and transitions $N_1 \rightarrow \cdots \rightarrow N_m$ and next taking a transition from node N_m to B. Rather than doing this, we will take a single *direct* transition from A to B whose cost will be the addition of costs of transitions $A \rightarrow N_1 \rightarrow \cdots \rightarrow N_m \rightarrow B$. Technically, no state or transition will be repeated in the working graph of RFD by taking this direct edge. Let us note that, if some states or transitions in the sequence $N_1 \rightarrow \cdots \rightarrow N_m$ were critical and unvisited, then we could take either this direct edge (and count traversed critical configurations) or some standard transitions to traverse the required configurations (combined with other direct edges to skip repeated parts). Let us note that there is no reason to take a direct edge more than once: since the goal of repeating nodes is reaching a *new* configuration, we can take the direct edge leading to that new configuration (which, obviously, has not been taken yet).

The previous idea must be refined to deal with some particular situations. In fact, a direct edge will *not* connect an origin state A with a destination state B. Instead, a different direct edge will be added to connect A with each (standard) *edge* leading to B. Thus, in our previous example, some direct edge will connect A with a node representing the *transition* connecting N_m and B (rather than directly connecting A with B). As we said before, edges are represented by barrier nodes. Thus, the direct edge will connect the origin edge with the *barrier node* representing such transition. The reason for this is the following: let us suppose that the edge connecting N_m with B is unvisited and critical, but N_m was visited before. This implies that the edge we used to leave N_m before was not the one leading to B. How can we reach and take the transition from N_m to B? On the one hand, since N_m has already been visited, taking the direct transition connecting A with N_m would imply following a *loop*, which is implicitly avoided by RFD. On the other hand, taking a direct transition from A to B would allow to implicitly cover the transition from N_m and B *only* if this transition were included in the shortest path from A to B. In

order to cover the transition from N_m to B without actually repeating N_m in our graph, we will use a direct edge from A to the *edge* between N_m and B. Let us note that having only this alternative notion of direct edge (that is, not having direct edges leading to *states*) does not disable our first example in the previous paragraph: if the goal is not covering the edge between N_m and B but covering B itself, then we can take the direct edge leading to the transition connecting N_m and B, and next move to B (the edge between N_m and B is necessarily unvisited; otherwise, B would have already been visited before).

In order to compute the cost of these additional edges, before launching RFD we will execute the *Floyd* algorithm for the graph representing our WFSM. Given a graph, the Floyd algorithm finds the shortest paths connecting each pair of nodes. After obtaining these shortest paths, for each pair of nodes A an B we will do as follows. Let us suppose that the (standard) transitions leading to B are $N_1 \xrightarrow{i_1/o_1/c_1} B, \ldots, N_m \xrightarrow{i_m/o_m/c_m} B$. Let us suppose that, according to Floyd algorithm, the shortest path from A to N_i has c'_i cost. Then, for all $1 \leq j \leq m$, the direct edge from A to the barrier node representing the transition $N_j \xrightarrow{i_j/o_j/c_j} B$ is given a cost $c'_j + c_j/2$. In addition, the transition connecting this barrier node with B will be given cost $c_j/2$. Thus, the total cost of moving from A to B through N_j will be $c'_j + c_j$, as expected.

Next we formally present the proposed graph transformation, and we prove the correctness of the approach.

Definition 5.1 Let $W = (S, s_{in}, I, O, C, \Delta)$ be a WFSM. The *shortcut machine* of W, denoted by $\texttt{shortcut}(W)$, is a WFSM $W' = (S', s_{in}, I', O', C, \Delta')$ where

- $S' = S \cup \Delta$, $I' = I \cup \{-\}$, and $O' = O \cup \{-\}$
- $\Delta' = \{(s, \delta, i, o, c/2) \mid \delta = (s, s', i, o, c) \in \Delta\} \cup$
  $\{(\delta, s', -, -, c/2) \mid \delta = (s, s', i, o, c) \in \Delta\} \cup$
  $\left\{(s, \delta, -, -, c + c_1/2) \;\middle|\; \begin{array}{l} s \in S, \delta = (s', s'', i, o, c_1) \in \Delta, \\ \text{the shortest path from } s \text{ to } s' \text{ has cost } c \end{array}\right\}$

□

Definition 5.2 Let $W = (S, s_{in}, I, O, C, \Delta)$, $S' \subseteq S$, and $\Delta' \subseteq \Delta$. Let $W' = \texttt{shortcut}(W)$ and $t' \in \texttt{Trees}(W')$. We say that t' *covers* S' (respectively, t' covers Δ') if for all $q \in S'$ (resp. $q \in \Delta'$) either q appears in t' or t' has a transition δ representing a shortest path α of W such that q is traversed by α. The maximal sets $S' \subseteq S$ and $\Delta' \subseteq \Delta$ such that t' covers S' and Δ' are denoted by $\texttt{StCoverShort}(t')$ and $\texttt{TrCoverShort}(t')$, respectively. □

Next we show that, given a machine W, searching for good trees where states and transitions are allowed to be repeated is equivalent to searching, in the machine $\texttt{shortcut}(W)$, for good trees where no state or transition is repeated. In particular, for all tree in W, we can find in $\texttt{shortcut}(W)$ a tree without repetitions that covers the same states and transitions and whose cost is equal to or lower than the cost of the former tree, that is, no relevant tree is lost by considering repetition-free trees in $\texttt{shortcut}(W)$. Besides, for all tree in $\texttt{shortcut}(S)$ that is free of repetitions we can find an equivalent tree in W with the same cost, that is, all trees in $\texttt{shortcut}(W)$ are possible in W. Thus, $\texttt{shortcut}(W)$ provides an appropriate working graph for applying RFD to solve our target problem. In the next result, let us note that numbers of *occurrences* refer to states and transitions of the *shortcut* machine. Thus, these numbers do not count the number of times we implicitly traverse states and transitions of the original machine by taking *direct* transitions, that is, by taking the transitions added to the shortcut machine to represent shortest paths.

Proposition 5.1 *Let* $W = (S, s_{in}, I, O, C, \Delta)$ *and* $W' = \texttt{shortcut}(W)$, *where* $W' = (S', s'_{in}, I', O', C', \Delta')$.

(a) *If* $t \in \texttt{Trees}(W)$ *then there exists* $t' \in \texttt{Trees}(W')$ *such that we have the properties* $\texttt{CostTree}(t) \geq \texttt{CostTree}(t')$, $\texttt{StateCoverT}(t) = \texttt{StCoverShort}(t')$, $\texttt{TranCoverT}(t) = \texttt{TrCoverShort}(t')$, *and for all* $s' \in S'$ (*respectively, for all* $\delta' \in \Delta'$) *the number of occurrences of* s' (*resp.* δ') *in* t' *is less than or equal to* 1.

(b) *If* $t' \in \texttt{Trees}(W')$ *and for all* $s' \in S'$ (*respectively, for all* $\delta' \in \Delta'$) *the number of occurrences of* s' (*resp.* δ') *in* t' *is less than or equal to* 1 *then there exists* $t \in \texttt{Trees}(W)$ *such that* $\texttt{CostTree}(t) = \texttt{CostTree}(t')$, $\texttt{StateCoverT}(t) = \texttt{StCoverShort}(t')$, $\texttt{TranCoverT}(t) = \texttt{TrCoverShort}(t')$. □

It is worth to point out that, in order to (implicitly) allow repetitions, adding new transitions directly connecting pairs of points through the shortest path is, in general, a better choice than adding several instances of each state in the graph. Let us note that if we want to reach X by traversing some repeated states then there is no reason for not taking the shortest path to X. Making an evolutionary computation method, such as RFD, find

these shortest paths by itself in a graph with several instances of each node is inefficient because the Floyd algorithm *optimally* solves this problem in polynomial time. In fact, adding direct transitions is a good choice unless repeating nodes is *rarely* preferable to loading, which happens only if the load cost C is very low. In this case, running the Floyd algorithm before executing RFD could be almost a waste of time because direct transitions would be rarely taken. Alternatively, we can bound the execution of the Floyd algorithm in such a way that we do not consider any direct edge whose cost is already known to be higher than C (in this case loading is better, so all direct transitions costing more than C can be ignored).

The second main modification of the general RFD scheme concerns load costs. The general RFD scheme does not include any negative incentive to form bifurcation points, i.e. points where two or more flows join together. However, a negative incentive must be included because these points will represent *loads* in our solutions, which imply some additional cost. Negative incentives should be proportional to the load cost, in such a way that loads are preferred only if repeating nodes is more expensive than loading. We consider the following incentive approach. Let us suppose that a drop can take an edge connecting its current node to a node N where some drop has already moved this turn. Note that moving to N would imply that the hypothetical solution formed by both drops would include a load at N. We will bias the *perception* of drops in such a way that, when the drop makes the (probabilistic) decision of where to move next, it will perceive as if the edge leading to N were C units higher than it actually is, where C is the load cost. Since the chances to take a path depend on the gradient, which in turn depends on the edge cost, this will reduce the probability to go to N. In fact, the drop will choose to go to N only if it is a good choice despite of the additional cost. Moreover, the erosion introduced after the drop moves to N, which in turn also depends on the edge gradient, will be calculated as if the edge cost were C units higher.

6. Experimental results

In this section we apply our method to empirically find good load trees for some `WFSMs`. Our experiments have two goals: (a) showing that being able to load previously traversed states may reduce the time needed to cover some states and transitions; and (b) showing that solutions provided by our heuristic method are good enough though not optimal. Regarding (a), we compare the time required to cover some critical configurations in the cases where load operations are allowed and not allowed. This comparison is made for three load cost assumptions: the load cost is similar to the cost of taking a few edges (so loading is usually preferable); the load cost is a bit less than the cost of the shortest path between two distant nodes (so loading is seldom preferable); and an intermediate point. In the alternative case where we cannot restore configurations, we assume that a *reliable reset button* is available, which is a typical assumption in testing methods. Thus, we actually assume we can only restore the initial state of the `WFSM`. An adapted version of RFD is also used to find solutions in this alternative case. Regarding (b), we compare the performance and optimality of results given by our method with those given by an optimal branch and bound (B&B) strategy.

Let us introduce the alternative problem we will use in this section for comparison purposes. This is the problem of covering a set of critical configurations when restoring previous configurations is not allowed, that is, when the only method to return to a previously traversed configuration is resetting the system and repeating the path from the initial state to this configuration. We assume that resetting has an associated cost. Since resetting implies putting some specific contents into the system memory (in particular, the contents of the initial configuration), in many practical cases the cost of resetting will be comparable with the cost of restoring a non-initial configuration in a environment where previously traversed configurations can be restored indeed. The problem of finding an optimal plan to test some target states/transitions when loading is not allowed but resetting (with an associated cost r) is formally defined next, and we prove that this problem is NP-complete too.

Definition 6.1 Let $W = (S, s_0, I, O, C, \Delta)$ be a `WFSM`. A *plain sequence* of W is a sequence $\sigma = s_1 \xrightarrow{i_1/o_1/c_1} s_2 \ldots s_{n-1} \xrightarrow{i_{n-1}/o_{n-1}/c_{n-1}} s_n$ where $s_1 = s_0$. The *cost* of the plain sequence σ, denoted by $\texttt{CostPlain}(\sigma)$, is equal to $\Sigma_{k=1}^{n-1} c_k$. Let $\alpha = \{\sigma_1, \ldots, \sigma_n\}$ be a set of plain sequences of W and $r \in \mathbb{N}$. The *resets-included cost of* α *for reset cost* r, denoted by $\texttt{ResetsIncludedCost}(\alpha, r)$ is equal to $\Sigma\{\texttt{CostPlain}(\sigma) \mid \sigma \in \alpha\} + (n-1) \cdot r$.

Given a WFSM $W = (S, s_0, I, O, C, \Delta)$, some sets $S' \subseteq S$ and $\Delta' \subseteq \Delta$, and two natural numbers $r, K \in \mathbb{N}$, the *minimum resets-included plan* problem, denoted by MRP, is stated as follows: is there a set α of plain sequences of W such that $\texttt{ResetsIncludedCost}(\alpha, r) \leq K$ and we have $S' \subseteq \bigcup_{\sigma \in \alpha} \texttt{StateCover}(\sigma)$ and $\Delta' \subseteq \bigcup_{\sigma \in \alpha} \texttt{TranCover}(\sigma)$. □

Theorem 6.1 MRP ∈ NP-complete. □

There exists some works where testing interactions with the minimum number of resets are constructed in the domain of FSMs [Hie04, HU10]. In [Hie04], given a set of sequences, a test sequence including all sequences from the set and having the minimum number of resets is composed. In [HU10], checking sequences (i.e. sequences allowing the tester to observe at least a fault in any faulty IUT having less than a given number of states) are constructed in such a way that they have the minimum number of resets. Let us note that the goal of problem MRP defined before is not minimizing the number of resets, but covering all states and transitions from a given set as cheaply as possible. Thus, depending on the cost of resetting (given by term r in Definition 6.1) and the cost of reaching each state without resetting, resetting might be a good choice or not in each case.

In order to adapt RFD to MRP, we apply the same ideas as before when we considered solving MLS by means of RFD. The main change with respect to the previous scheme consists in modifying the negative incentive to form bifurcations points. Let us note that, in this case, the additional cost due to including a bifurcation point does not consist only of the reset cost itself, but also of the cost to go from the initial state (the sea) to the bifurcation point. Note that a reset operation does not recover the bifurcation point but the initial state, so next we have to reach the bifurcation point again. The negative incentive to make a bifurcation point will be proportional to the sum of the reset cost and the cost of going from the sea to the node where the bifurcation is formed. Let us suppose that a drop is in node N_1, there is an edge between N_1 and N_2, and at least one drop has moved to N_2 in this turn. Then, the probability of this drop to go to N_2 is proportional to the difference of altitudes and inverse proportional to the sum of the cost of the edge connecting N_1 and N_2, the reset cost r, and the cost of the direct edge connecting the sea to N_2 (this value is 0 if N_2 is the sea). Let us note that, in practice, this negative incentive will avoid the formation of bifurcation points that are *far* from the sea (which are the most expensive ones to reach after a reset operation). Thus, formed solution trees will tend to have their bifurcation points near to the root (i.e. near to the initial state).

After executing RFD for solving MLS or MRP, we construct the solution tree as follows. If the set of *critical* edges, that is the set of edges required to be covered in our problem instance, is not empty, then we start by selecting one of these critical edges. Next we follow, from this edge, the best slopes (i.e. the ones providing highest down gradients) for going down to the sea, and we introduce in the tree all nodes and edges visited in the way. Then we take another critical edge that is not in the constructed tree yet, and we follow the best slopes down to the sea or to a node/edge already included in the solution tree, whatever comes first, including in the tree all traversed edges and nodes. The process is repeated until all critical edges are included in the solution tree, and next we follow the same process for *critical* nodes, that is, nodes required to be covered in our problem instance, from the highest critical node to the lowest one.

The previous tree construction scheme is modified to improve the quality of formed solutions as follows. Let us suppose that, by following the best down slopes to the sea from some critical node/edge, we reach some node s_i and the best available edge from s_i (i.e. the one with the highest down gradient) is an edge $s_i \rightarrow s_j$ leading to a node s_j already included in the tree. If $s_i \rightarrow s_j$ is added to the tree indeed, then a bifurcation point will be introduced, so an additional load or reset operation will be added to the solution in MLS or MRP, respectively. In these cases, we will check if it is cheaper creating that load/reset (that is, adding $s_i \rightarrow s_j$) or trying to avoid this load/reset in two possible ways: (i) joining s_i to some *leaf* of the current tree by taking some direct transition to the leaf; or (ii) adding $s_i \rightarrow s_j$ indeed, but modifying the current tree so that the transition already existing in the tree that leads to s_j, say $s_k \rightarrow s_j$, is substituted by a direct transition from s_k to the leaf of the new branch (i.e. the node where we started the branch). There exists a particularity in the case of MRP. If we observe that avoiding the reset as said in (i) and (ii) is not better than resetting, then we will not add $s_i \rightarrow s_j$ to the tree, but we will add the *direct transition* leading to s_0, the initial state of the FSM. In this way, the testing path will reach s_i from s_0 through the shortest path.

On the other hand, the B&B algorithm used for solving MLS works as follows. First, we create an initial and trivial solution. This solution consists in joining all critical nodes/edges directly with the sea (the initial state) by using the shortest path. In the first call to the recursive function, the initial state of the FSM is set as the root of the solution tree. The parameters of the recursive call are the following: nodes and edges already visited, nodes

that are leaves in the partial solution (used to control needed loads), the cost of the partial solution constructed so far (from now on, denoted by $currentCost$), and the number of loads. We extend the partial solution by adding some edge connected with the formed solution that does not belong to the solution yet, and we recursively call the function again with the new partial solution. The recursion finishes when all critical nodes/edges are included in the tree. Besides, in order to prune some useless alternatives, we stop the development of those partial solutions such that $currentCost + minCost$ is higher than the best solution found so far, where $minCost$ is the sum of the costs of all critical edges not included in the solution yet. Clearly, $minCost$ is a lower bound of the cost of adding all critical nodes/edges not visited yet to the solution tree.

Next we present our experimental results. All experiments were performed using an Intel T2400 processor with 1.83 GHz. RFD was executed fifty times for each of the graphs during 5 min, while the B&B method was executed once and given 2 h (note that B&B is deterministic, so running it more times is pointless). For each method, Table 1 summarizes the best solution found (Best), the arithmetic mean (Avg), the best solution found for the *reset case*, that is the alternative case where we cannot load any state different to the initial state (Rst best), and the arithmetic mean in this case (Rst avg).

In Table 1, symbol '–' denotes a non-applicable case. The inputs of both algorithms are randomly generated graphs with 50, 100 and 200 nodes where the cost of edges is between 0 and 100. In the case that the graphs are sparse (✓), each node is connected with 2–5 nodes. However, when the graphs are dense (×), each node is connected with 80% of the nodes. We present the results when the load/reset cost is relatively cheap (20), when the load/reset cost is medium (100) and when the load/reset cost is high (1,000) with respect to the cost of an edge of the `WFSM`.

As we can see in Table 1, being able to load previously traversed states reduces the time needed to cover all the critical points in sparse graphs (see columns *Best* and *Rst best*). Let us note that in the case of dense graphs it is relatively easy to find paths covering all the critical points without requiring any load or reset. Thus, columns *Best* and *Rst best* are nearly equal. However, in the case of sparse graphs (in fact, the most typical case of FSM specification) the advantages are obvious: in nearly all cases the column *Best* outperforms the column *Rst best*. As it can be expected, in the cases where the load cost is not very high, the difference is even bigger. Thus, cheaper testing plans can be obtained when load operations are available.

Regarding the quality of the solutions found by RFD, we must analyze the differences between *RFD* and *B&B* rows. Let us recall that RFD was executed for only 5 min while the B&B strategy was running for 2 h. Those cases marked with an asterisk (*) denote that the B&B algorithm did not improve the initial solution during the 2 h execution. Anyway, the results obtained by RFD are always better than those of B&B. In fact, though B&B can eventually find the optimal solution, the performance of RFD and B&B for similar execution times is incomparable: even if we execute B&B for much longer times than RFD, solutions found by RFD clearly outperform those given by B&B. So, executing RFD helps us to create test plans for testing an application and to reduce the time needed for this task.

7. Case study

In this section we apply our testing methodology to a simple case study. We test a social network management application. This application lets the system manager check and modify data from the network users. In this example we will be concerned with the following functionalities: adding a user, deleting a user, and modifying the friends of a user in some predefined ways. For each user we will be concerned with only two data: her complete name (which must be fixed and unique) and her list of friends. Thus, in this example, modifying a user will consist just in modifying her list of friends. Friends of a user can be modified either manually (i.e. the manager directly selects the friends to be added/deleted) or according to a given *friendship operation*. The first friendship operation consists in modifying the friends set of a user by adding all users having a friendship relation with any current friend of the user (i.e. *my friend's friends are my friends*). On the other hand, given a user and a *target new friend*, the second friendship operation also adds some users to the user's friends set, in particular all users in the *shortest path* of friendship links from the user to the target new friend (i.e. *those people most directly leading to my new friend are my friends*). Besides, in order to ban users aiming at using the social network for commercial purposes (e.g. to contact potential customers), the system will not allow any user U to have more than 50 *completely isolated friends*, i.e. friends that do not have friendship with any other friend of U. In particular, the system will not allow the creation of a user not fulfilling this condition. Beyond these specific functionalities, the system will be required to fulfill the expected requirements in a correct database management system (such as, e.g. a user should not disappear from the system unless the manager deletes it), as well as other specific conditions. Let us

Table 1. Summary of results of experiments presented in Sect. 6 (up to the 200 nodes graph) and Sect. 7 (the 252 nodes graph)

Method	Graph size	Sparse	Load cost	Best	Avg	Rst best	Rst avg
RFD	50	✓	20	1,327	1,379	1,873	1,982
B&B	50	✓	20	3,213	–	–	–
RFD	50	✓	100	1,935	2,193	2,249	2,317
B&B	50	✓	100	5,453	–	–	–
RFD	50	✓	1,000	4,047	4,123	4,079	4,147
B&B	50	✓	1,000	30,653	–	–	–
RFD	100	✓	20	3,979	4,154	4,615	4,944
B&B	100	✓	20	6152	–	–	–
RFD	100	✓	100	4,472	4,729	4,863	5,205
B&B	100	✓	100	10,312	–	–	–
RFD	100	✓	1,000	6,275	6,576	6,618	6,935
B&B	100	✓	1,000	57,112	–	–	–
RFD	200	✓	20	8,544	9,244	9,250	9,881
B&B	200	✓	20	20,570*	–	–	–
RFD	200	✓	100	9,796	10,631	9,764	10,740
B&B	200	✓	100	27,370*	–	–	–
RFD	200	✓	1,000	12,418	13,374	12,675	13,649
B&B	200	✓	1,000	103,870*	–	–	–
RFD	50	×	20	811	823	816	838
B&B	50	×	20	3,974	–	–	–
RFD	50	×	100	806	890	810	895
B&B	50	×	100	7,654	–	–	–
RFD	50	×	1,000	827	1,427	808	1,488
B&B	50	×	1,000	49054	–	–	–
RFD	100	×	20	1,242	1,271	1,237	1,282
B&B	100	×	20	2,309*	–	–	–
RFD	100	×	100	1,249	1,287	1,253	1,325
B&B	100	×	100	5,909*	–	–	–
RFD	100	×	1,000	1,213	1,527	1,231	1,776
B&B	100	×	1,000	46,409*	–	–	–
RFD	200	×	20	1,129	1,136	1,110	1,126
B&B	200	×	20	2,829*	–	–	–
RFD	200	×	100	1,285	1,295	1,279	1,288
B&B	200	×	100	9629*	–	–	–
RFD	200	×	1,000	3,069	3,095	3,076	3,088
B&B	200	×	1,000	86,129*	–	–	–
RFD	252	Case study	20	1,620	1,695	1,787	1,873
B&B	252	Case study	20	6,880*	–	–	–
RFD	252	Case study	100	2,123	2,165	2,243	2,320
B&B	252	Case study	100	12,080*	–	–	–
RFD	252	Case study	1,000	7,525	7,585	7,685	7,741
B&B	252	Case study	1,000	70,580*	–	–	–

denote a system user U by a pair $U = (name, friends)$ where $name$ is the name of U and $friends$ is the set of names of the friends of U. We will assume that friendship is a symmetric relation in our system, i.e. given two users $U = (name, friends)$ and $U' = (name', friends')$, we have $name \in friends'$ iff $name' \in friends$. So, adding $name$ to $friends'$ should immediately add $name'$ to $friends$ (otherwise we would have a system failure). Moreover, we also assume that friendship is anti-reflexive (i.e. $name \notin friends$) because it would not add valuable information. Besides, if the manager deletes a user then the system will be required to remove the user from any friend set of any user, and the system will not allow the addition of a user with an already existing name.

In order to check the functionality of this application under typical run-time conditions, our testing activity will start from a configuration where there are one million randomly generated users in the system database. Obviously, checking any possible configuration of a system with one million users is unfeasible. Instead, we will concentrate on exhaustively studying the system behavior for all possible configuration of *three* arbitrary users belonging to the set of users. Let X_1, X_2, X_3 be three users belonging to the system at the initial configuration. A user X_i with $1 \leq i \leq 3$ will be denoted by a tuple $X_i = (name_i, friends_i)$ where $name_i$ is the name of X_i and $friends_i$ is the set of names of the friends of X_i. We will add a 0 superfix to denote the values of these data at the initial configuration, i.e. $X_i^0 = (name_i^0, friends_i^0)$ denotes the data stored for X_i at the beginning of the experiment. In order to distinguish X_1, X_2, X_3 from the rest of users in the system, we will call these users *control users*. We will assume that $name_i^0 \notin friends_j^0$ for all $i \neq j$ with $i, j \in \{1, 2, 3\}$, that is, our control users are not friends in the initial configuration. Our testing plan will consist of observing all possible behaviors of the system at all possible configurations *up to control users*. We consider that two system configurations are equivalent up to control users if they contain the same information regarding the (non-)existence of X_1, X_2, X_3 and the (non-) presence of these users in each other's friends sets. For each configuration c and for each operation o involving the creation, deletion, or modification of users X_1, X_2, or X_3, our testing plan will aim at observing at least once the operation o from some configuration being equivalent to c. In our experiments, we will assume that if X_i is deleted and re-created later, then it will be given again the initial value $X_i^0 = (name_i^0, friends_i^0)$.

The expected functionality of the system is depicted in Fig. 2 in the form of an *extended* finite state machine (EFSM). An EFSM is an FSM where the configuration of a system depends not only on the current state but also on the current values of a set of *variables*. In our example, these variables will be the current values of X_1, X_2, and X_3. An EFSM transition is labeled by an input and an output, like any FSM, together with some new information: a boolean guard on the variables (which must be true to trigger the transition) and an action on the variables (which determines how variables change after taking the transition). If the domain of variable values is finite, an EFSM can be transformed into an equivalent FSM by expanding it into any combination of $(state, values\ of\ variables)$. Since our experiment will only concern the creation/deletion/modification/observation of control users X_1, X_2, X_3, the set of possible values of variables consists of all combination of existence/inexistence of each of these users, as well as all combination of friend sets of each (existing) control user, where combinations differ only in X_1, X_2, X_3. In particular, the (non-)presence of non-control users will be ignored and will not motivate the existence of additional variable values. It is easy to see that, according to the restrictions imposed to a (correct) system, only 18 combinations of variable values are possible in the EFSM specification.[4]

The central state of the EFSM depicted in Fig. 2, HOME, is the state where the system manager chooses what to do next. We can choose to add/delete a user (ADD USER, DEL USER), to add/delete a friend to the friends set of some user (ADD FRIEND, DEL FRIEND), or, given a user, we can add all friends of her current friends to his set of friends (ADD FR OF FR) or we can add to her list of friends all friends in the shortest path to some target new friend (ADD FR TO X). After selecting the operation to be performed, we reach a state where we can either cancel the operation or specify the target users of the operation. If correct target users are given, the system reaches a new state denoting that the operation was correctly performed (the one whose name finishes with 'OK'). Besides, from the HOME state we can also go to the QUERIES state, where we can check whether some user exists or not, as well as the set of friends of a user. The definition of each transition depicted in Fig. 2 is given in Table 2. Transitions follow the form "[*guard on variables*] *input* | *output* [*action on variables*]".

[4] We count as follows: one combination where none of the control users is present in the system, three combinations with only one control user (which cannot be friend of the others), three combinations with two control users (where these two users can be friends or not, leading to 6 combinations indeed), and 1 combination with all three control users, which splits into the following possibilities: one combination where none is friend of any other, three combinations where there is only one friendship link out of the three available ones, three combinations where there are two friendship links, and 1 combination where all three available friendship links exist.

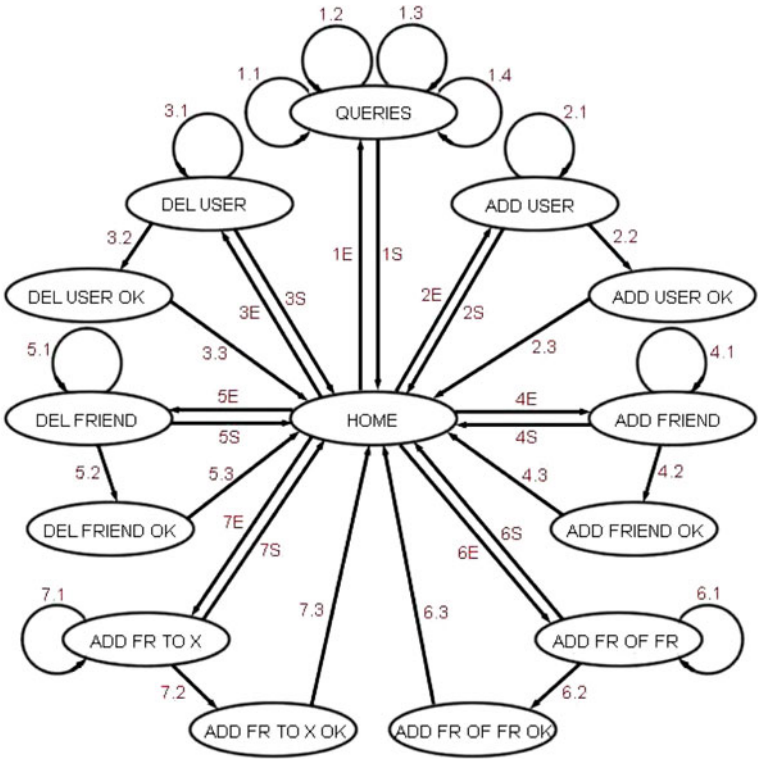

Fig. 2. Specification of the social network manager system

In transitions where the input consists in introducing a user's *nick*, we will consider that this nick must be $name_1^0$, $name_2^0$, or $name_3^0$, i.e. a name of a control user. So, in this case the EFSM provides a compact notation to denote three standard FSM transitions. Similarly, in transitions where the input also includes a friend of a set of friends, a control user or a set of control users will be given, respectively. The meaning of functions used in guards and actions of transitions given in Table 2 is straightforward.

The goal of our testing methodology will be to traverse at least once all critical configurations introduced by the user of FSM derived from the original EFSM in the shortest possible time. Since the EFSM has 14 states, the derived FSM has $14 \times 18 = 252$ states.[5] In order to decorate FSM transitions with their associated time consumption, we assign time delays to system operations as follows. The cost of remotely accessing a database with one million registers may be relatively high for some operations, in particular for those where more than one register has to be checked or modified. For instance, let us note that adding a user $(nick, friends)$ implies checking that the number of completely isolated friends in the set $friends$ (i.e. the number of users in $friends$ that do not have any friend belonging to $friends$) is under 50, which in turn implies checking the sets of friends of all users in $friends$. Thus, many registers might have to be checked. We consider that (correctly) adding a user to the database takes 25 s, which is also the cost of detecting an error in this operation. Adding a friend to some user's friends set takes 12 s, while deleting a user or a friend from some user's friends set takes 10 s. Other operations are more complex and have higher costs, such as adding the friends of current friends of some user (which costs

[5] Actually, there are a few less states because some of these 252 states are not reachable. For instance, it is not possible to reach an EFSM configuration where the state is ADD USER OK and none of X_1, X_2, X_3 are present in the database.

60 s) and adding the friends in the shortest path to a target new friend (which costs 60 s as well). In these cases, detecting an error takes 60 s as well. We consider that all query operations require a short time, in particular 10 s. The transitions from/to the HOME state are set to 1 s. Finally, the time to recover a previously traversed configuration, that is the time required to load a one million registers database, is given a different value in each experiment, as we did in the experiments of the previous section. In particular, times 20, 100, and 1,000 are considered.[6] Let us note that the system state basically consists in the state of the database, because the memory required to store the system interface state is negligible in comparison with the database size.

Table 2. Transitions of the EFSM

1E.	[ ] pushBtnQueries | screenQueries [ ]
1S.	[ ] pushBtnReturnQueries | screenHome [ ]
1.1	[not exists(nick, BD)] exists(nick)? | "The user does not exist." [ ]
1.2	[exists(nick, BD)] exists(nick)? | "The user exists." [ ]
1.3	[not exists(nick, BD)] friends(nick)? | "The user does not exist." [ ]
1.4	[exists(nick, BD)] friends(nick)? | "The friends are:" ++ friends(nick,BD) [ ]
2E.	[ ] pushBtnAddUser | screenAddUser [ ]
2S.	[ ] pushBtnReturnAddUser | screenHome [ ]
2.1	[not validAddUser(nick, friends, BD)] (nick, friends) | "The user already exists or has too many completely isolated friends." [ ]
2.2	[validAddUser(nick, friends, BD)] (nick, friends) | "New user added." [addUser(nick, friends, BD)]
2.3	[ ] pushBtnReturnAddUserOk | screenHome [ ]
3E.	[ ] pushBtnDelUser | screenDelUser [ ]
3S.	[ ] pushBtnReturnDelUser | screenHome [ ]
3.1	[not validDelUser(nick, BD)] (nick) | "The user does not exists." [ ]
3.2	[validDelUser(nick, BD)] (nick) | "The user has been deleted." [delUser(nick, BD)]
3.3	[ ] pushBtnReturnDelUserOk | screenHome [ ]
4E.	[ ] pushBtnAddFriend | screenAddFriend [ ]
4S.	[ ] pushBtnReturnAddFriend | screenHome [ ]
4.1	[not validAddFriend(name, friend, BD)] (name, friend) | "The friend already exists." [ ]
4.2	[validAddFriend(name, friend, BD)] (name, friend) | "New friend added." [addFriend(name, friend, BD)]
4.3	[ ] pushBtnReturnAddFriendOk | screenHome [ ]
5E.	[ ] pushBtnDelFriend | screenDelFriend [ ]
5S.	[ ] pushBtnReturnDelFriend | screenHome [ ]
5.1	[not validDelFriend(name, friend, BD)] (name, friend) | "The friend does not exists." [ ]
5.2	[validDelFriend(name, friend, BD)] (name, friend) | "The friend has been deleted." [delFriend(name, friend, BD)]
5.3	[ ] pushBtnReturnDelFriendOk | screenHome [ ]
6E.	[ ] pushBtnFriendsOfMyFriends | screenFriendsOfMyFriends [ ]
6S.	[ ] pushBtnReturnFriendsOfMyFriends | screenHome [ ]
6.1	[not existsFriendsOfMyFriends(name, BD)] (name) | "There are not new friends." [ ]
6.2	[existsFriendsOfMyFriends(name, BD)] (name) | "Friends of friends have been added." [addFriendsOfMyFriends(name, BD)]
6.3	[ ] pushBtnReturnFriendsOfMyFriendsOk | screenHome [ ]
7E.	[ ] pushBtnAddFriendToX | screenAddFriendX [ ]
7S.	[ ] pushBtnReturnAddFriendToX | screenHome [ ]
7.1	[not existsChainFriendsToX(name, friendX, BD)] (name, friendX) | "Chain to friendX cannot be added." [ ]
7.2	[existsChainFriendsToX(name, friendX, BD)] (name, friendX) | "Chain to friendX added." [addChainFriendsToX(name, friendX, BD)]
7.3	[ ] pushBtnReturnAddFriendToXOk | screenHome [ ]

[6] In practice, 100 s could be a feasible time for making a copy of a database with one million registers, as its actual size could be a few gigabytes—assuming it contains data other than just user names and friends sets.

We conducted experiments to find good testing plans for this problem by following the same methodology as in the previous section. Regarding the *reset case*, i.e. the case where only the initial state can be recovered, we consider the same possible costs as in the case where any previously traversed state can be restored (that is, 20, 100, and 1,000 s). Experimental results are depicted at the bottom of Table 1. As we can see, RFD obtains again better results than B&B. Besides, being able to restore any previously traversed state is again better than only being able to reset, though the advantage is reduced in this case. Let us note that, in this system, the average cost to reach a state from another state is not very high because there do not exist two states that are very distant from each other. For instance, given a configuration where none of X_1, X_2, and X_3 exist, reaching a new configuration where X_1, X_2, X_3 exist and all of them are friends of each other requires only six database operations. Let us recall that the initial configuration is a state where X_1, X_2, and X_3 exist but none of them is friend of each other. Thus, it is easy to see that reaching any state from the initial state requires at most three database operations, which is a very low number of transitions for a 252 states FSM. In particular, coming back to a previously traversed state by resetting and repeating some path is relatively *cheap* in all cases because the path to be repeated is short. It is remarkable that, even in such an adverse scenario for a general loading framework, being able to load any configuration is still (a bit) better than just resetting.

8. Conclusions and future work

In this paper we have presented and studied a testing scenario where we can copy and restore previously traversed configurations of the IUT at some cost. Being able to copy and restore complete state representations of the IUT is generally possible when the IUT is a software system. It may also be possible in a hardware system, provided that the complete state memory of the device can be dumped and reinserted later. Beyond these basic requirements, the practical applicability of such feature requires that, in addition, operations performed by the IUT are time-expensive but the memory used by it is, in comparison, not so high. As we have shown in our case study, this restriction can be fulfilled even by a system being as memory-intensive as a database management system, provided that some data manipulation operations require high times. In this case, using load/restore operations allows a reduction in the time needed to cover some given critical IUT configurations, because loading will often be cheaper than repeating some already traversed path. We have shown that finding optimal plans to interact with the IUT for this particular testing case is an NP-hard problem, so an heuristic approach is required to find good test sequences (in particular, good test *trees*). We have applied an evolutionary computation approach to solve this problem, in particular *river formation dynamics*. Experimental results show that (a) loading is a good choice even in scenarios where the cost of loading is not particularly low; and (b) the RFD implementation provides a good tradeoff between (partial) optimality of results and the time needed to achieve them.

As future work, we wish to apply another evolutionary computation approach, *ant colony optimization*, to this problem. RFD and ACO have already been compared in the context of other NP-hard problems, but the performance of both methods to solve `MLS/MRP` should be studied. Previous studies show that RFD fits particularly well for problems consisting in forming a kind of covering tree [RRR08, RRR09a], though an hybrid RFD-ACO approach could outperform the results provided by each method if we were able to take the best parts of both methods. Besides, we wish to generalize our method to the case where *extended* finite state machines are considered. We have dealt with an EFSM in the case study of this paper, but we have done it by expanding it into an equivalent FSM. However, RFD adaptations considered in [RRR09a] show that RFD can deal with *variables* without necessarily unfolding FSM states into all combinations of (*variable value, state*). Thus, the capability of our method to directly deal with EFSMs, without expanding them into FSMs, should be studied.

A. Proofs

A.1. Proof of Theorem 3.1

Given a load sequence σ, we can easily compute $\texttt{CostSeq}(\sigma)$ (and compare it with K), as well as $\texttt{StateCover}(\sigma)$ and $\texttt{TranCover}(\sigma)$ (and compare them with S' and Δ', respectively), in polynomial time with respect to W, S', Δ', and K, so we have $\texttt{MLS} \in \texttt{NP}$.

In order to prove MLS $\in$ NP-complete, we polynomially reduce an NP-complete problem to MLS. Let us consider the *Hamiltonian Path problem* in directed graphs, which is stated as follows: given a directed graph $\mathcal{G}$ (without costs associated with transitions), find a path traversing all nodes exactly once. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (where $\mathcal{V}$ and $\mathcal{E}$ denote the sets of vertices and edges, respectively, and we suppose $n =\mid \mathcal{V} \mid$) represents an instance of the Hamiltonian Path problem. We construct n instances of MLS from $\mathcal{G}$ as follows. Let $W_1, \ldots, W_n$ be WFSMs where, for all $1 \leq i \leq n$, we have $W_i = (S, s_i, I, O, C, \Delta)$ (note that, for all W_i, we have the same values for all tuple components but the initial state, s_i). Sets S and Δ, that is, sets of states and transitions of *all* considered WFSMs, literally copy the form defined by $\mathcal{V}$ and $\mathcal{E}$ in $\mathcal{G}$. In addition, inputs and outputs labeling transitions in Δ are defined in such a way that the resulting WFSM is *deterministic*: we consider $I = \{i_1, \ldots, i_n\}$ for some input symbols $i_1, \ldots, i_n$, $O = \{o_1, \ldots, i_n\}$ for some output symbols $o_1, \ldots, o_n$, and, for all $s \in S$, all transitions leaving s are labeled by a *different* input/output pair i_j/o_j. The cost of all transition $\delta \in \Delta$ is set to 1, and the load cost C is equal to n. For each W_i, its initial state s_i is set to a different state in S (thus, for all $s \in S$ there is some $1 \leq j \leq n$ such that the initial state of W_j is s). In addition, let $S' \subseteq S$ and $\Delta' \subseteq \Delta$ be defined as $S' = S$ and $\Delta' = \emptyset$, and let $K = n - 1$. Since each W_i can be constructed from $\mathcal{G}$ in polynomial time, constructing MLS problem instances $(W_1, S', \Delta', K), \ldots, (W_n, S', \Delta', K)$ from $\mathcal{G}$ requires polynomial time.

Let us see that the answer of the Hamiltonian Path problem for $\mathcal{G}$ is yes iff there exists $1 \leq i \leq n$ such that the answer of MLS to W_i, S', Δ', K is yes. Let us consider the implication from left to right. Let σ be a Hamiltonian path for $\mathcal{G}$ and W_i be the WFSM such that its initial state is the initial state of σ. We have that σ is a load sequence for W_i, σ covers all states and transitions in S' and Δ', and $\texttt{CostSeq}(\sigma) = n - 1 = K$. Now, let us consider the implication from right to left. If (a) σ is a load sequence for some W_i; (b) σ covers all states and transitions in S' and Δ'; and (c) $\texttt{CostSeq}(\sigma) \leq K = n - 1$, then there is no *load* in σ (otherwise we would have $\texttt{CostSeq}(\sigma) > K$). Since all n states of S' are covered by σ, σ must take at least $n - 1$ transitions. Note that the cost of all transitions is 1. Thus, $\texttt{CostSeq}(\sigma) \leq n - 1$ implies that σ traverses exactly $n - 1$ transitions. If σ traverses n different states in $n - 1$ transitions then no state is traversed more than once in σ. Thus, σ is a Hamiltonian path for $\mathcal{G}$.

A.2. Proof of Proposition 3.1

Let $\sigma = \sigma_1 \cdot \cdots \cdot \sigma_n$ be such that for all $1 \leq j \leq n$ we have $\sigma_j = s_{1,j} \xrightarrow{\delta_{1,j}} \cdots \xrightarrow{\delta_{k_j,j}} s_{k_j,j}$, where $\delta_{k_j,j} = \psi(s_{k_j,j})$ and for all $1 \leq l < k_j$ we have $\delta_{l,j} = i_{l,j}/o_{l,j}/c_{l,j}$ for some $i_{l,j}$, $o_{l,j}$, $c_{l,j}$. Let us iteratively construct a sequence σ' from σ as given in the following algorithm. Given a load sequence σ, this algorithm constructs a new load sequence σ' by splitting σ into all possible subsequences where a load is executed at the last step, and next these subsequences are re-joined in such a way that loads refer only to the states traversed most recently. We will see that, by doing so, the returned load sequence σ' will be an α-load sequence. Moreover, σ' will traverse the same states/transitions as σ and will make the same number of load operations, so both sequences will have the same cost.

$R := \{(s_{1,j}, (\sigma_j)^-) \mid 2 \leq j \leq n\}$;
$\sigma' := (\sigma_1)^-$;
$stack :=$ empty stack;
push onto $stack$, in appearance order, all states appearing in $(\sigma_1)^-$;
while $R \neq \emptyset$ **do**
  $first := top(stack)$;
  while $\nexists\ \sigma''$ such that $(first, \sigma'') \in R$ **do**
    $stack := pop(stack)$;
    $first := top(stack)$;
  od
  $\sigma'' :=$ any σ'' such that $(first, \sigma'') \in R$;
  $R := R \backslash (first, \sigma'')$;
  $last := lastStateOf(\sigma')$;
  $\sigma' := \sigma' \cdot (last \xrightarrow{\psi(first)} first) \cdot \sigma''$;
  push onto $stack$ all states in σ'' that are not already included in $stack$;
od
return σ';

Let us see that (a) the previous algorithm always terminates; and (b) σ' is such that $\sigma' \in \alpha\texttt{-Sequences}(W)$ and we have $\texttt{CostSeq}(\sigma) = \texttt{CostSeq}(\sigma')$, $\texttt{StateCover}(\sigma) = \texttt{StateCover}(\sigma')$, and we also have $\texttt{TranCover}(\sigma) = \texttt{TranCover}(\sigma')$. In order to check (a), let us see that, for each round of the outer `while`, an element is taken away from R, and R never increases. Thus, as long as the inner `while` always terminates, the outer one will always do so. We consider the inner `while`. Let us check that, by popping elements from $stack$, we will eventually find some state $first$ such that, for some σ'', we have $(first, \sigma'') \in R$. By contradiction, let us suppose that $stack$ is fully emptied without finding such state $first$. This means that there is no pair (s, σ'') in R such that $s \in stack$. For all $(s, \sigma'') \in R$, either s is traversed in the current value of σ' or not. If it does then, since all states traversed by σ' are eventually introduced in $stack$ but we have $s \notin stack$, we have that s has been *removed* from $stack$ before. By the construction of $stack$, s is removed from $stack$ only after there are no more elements (s, σ'') in R. Thus, for all $(s, \sigma'') \in R$, we have that s has not been traversed yet by σ'. Let R^1 be the value initially given to R in the algorithm. Since σ is the result of extending $(\sigma_1)^-$ with the concatenation of all subsequences σ'' such that $(s, \sigma'') \in R^1$ with appropriate loads to previously traversed states, we infer that for all $(s, \sigma'') \in R^1$ either there exists $(s', \sigma''') \in R^1$ such that s is traversed in σ''' or s is traversed by $(\sigma_1)^-$. Since for all $(s, \sigma'') \in R \subseteq R^1$ we have that s has not been traversed yet by σ', we infer that, for all $(s, \sigma'') \in R$, s is not traversed by $(\sigma_1)^-$ and all pairs $(s', \sigma''') \in R^1$ such that s is traversed in σ''' are such that $(s', \sigma''') \in R$ (otherwise we would have $(s', \sigma''') \in R^1 \backslash R$ and s would be traversed by σ'). If states appearing in the first element of pairs belonging to $R \subseteq R^1$ are traversed only in subsequences referred in the second element of pairs belonging to $R \subseteq R^1$ then σ cannot be the result of extending $(\sigma_1)^-$ with a concatenation of all subsequences σ'' such that $(s, \sigma'') \in R^1$ with appropriate loads: connecting subsequences σ'' with $(s, \sigma'') \in R$ to the rest of the sequence would not be possible. Hence, we have a contradiction.

Next we consider (b). Let $\sigma' = s_1 \xrightarrow{\delta_1} s_2 \cdots s_{n-1} \xrightarrow{\delta_{n-1}} s_m$ be the sequence returned after terminating the algorithm. Loads introduced in σ' are forced to refer to states that are as *upper* in $stack$ as possible. Thus, no load to a previously traversed state s is introduced in σ' until all required loads leading to states traversed *after* s are introduced. Thus, there cannot exist states s_i, s_j with $1 \leq i < j \leq m$ such that for all $k < i$ we have $s_k \neq s_i$, for all $l < j$ we have $s_l \neq s_j$, and there exist $j < p < q \leq n$ such that $\delta_p = \psi(s_i)$ and $\delta_q = \psi(s_j)$. Hence, $\sigma' \in \alpha\texttt{-Sequences}(W)$. Besides, let us note that σ' includes exactly the same transitions of σ, as well as the same number of loads. Thus, $\texttt{CostSeq}(\sigma) = \texttt{CostSeq}(\sigma')$ and $\texttt{TranCover}(\sigma) = \texttt{TranCover}(\sigma')$. Moreover, the same states are traversed in σ and σ', so we have $\texttt{StateCover}(\sigma) = \texttt{StateCover}(\sigma')$.

A.3. Proof of Proposition 3.2

Let us consider (a). In particular, we show that for all $\sigma \in \texttt{createSeq}(t)$ we have $\sigma \rhd t$. We prove it by induction over the structure of t. As anchor case, let us consider $t = (root, [\,])$. In this case, $\texttt{createSeq}(t)$ contains a single element $\sigma = root$. Let us note that $\sigma \rhd t$ means $\texttt{createTree}(\sigma) \equiv_T t$. We have $\texttt{createTree}(\sigma) = (root, [\,])$, and trivially we conclude $t \equiv_T (root, [\,])$. Let us consider the inductive case, that is, $t = (root, [(tr_1, child_1), \ldots, (tr_n, child_n)])$ with $n \geq 1$. In this case, for all $\sigma \in \texttt{createSeq}(t)$ we have that σ is the result of concatenating subsequences $root \xrightarrow{tr_i} \sigma'_i \xrightarrow{\psi(root)} root$ in any order, where $\sigma'_i \in \texttt{createSeq}(child_i)$, and next removing the last step. Let us show that $\sigma \rhd t$, that is, $\texttt{createTree}(\sigma) \equiv_T t$. We have $\texttt{createTree}(\sigma) = (root, l)$, where l is the result of concatenating the pairs $(tr_1, \texttt{createTree}(\sigma'_1)), \ldots, (tr_n, \texttt{createTree}(\sigma'_n))$ in any order. Thus, conditions (1) and (2) required by the $\equiv_T$ relation (see Definition 3.7) hold in $\texttt{createTree}(\sigma) \equiv_T t$. Let us match pairs $(tr_1, child_1), \ldots, (tr_n, child_n)$ with pairs $(tr_1, \texttt{createTree}(\sigma'_1)), \ldots, (tr_n, \texttt{createTree}(\sigma'_n))$, respectively. The transition tr_i coincides for each match. Let us check that the pair of trees in each match are equivalent, that is, that for all $1 \leq i \leq n$ we have $child_i \equiv_T \texttt{createTree}(\sigma'_i))$. By induction hypothesis, let us suppose that for all $1 \leq i \leq n$ we have that $\texttt{createSeq}(child_i) = \{\sigma' \mid \sigma' \rhd child_i\}$. This implies that for all $\sigma' \in \texttt{createSeq}(child_i)$ we have $\texttt{createTree}(\sigma') \equiv_T child_i$. Since for all $1 \leq i \leq n$ we have $\sigma'_i \in \texttt{createSeq}(child_i)$, we conclude $child_i \equiv_T \texttt{createTree}(\sigma'_i)$. Thus, condition (3) required by $\equiv_T$ is met and we prove that for all $\sigma \in \texttt{createSeq}(t)$ we have $\sigma \rhd t$. Proving the opposite set inclusion, that is, that for all σ with $\sigma \rhd t$ we have $\sigma \in \texttt{createSeq}(t)$, is similar.

In order to prove (b), we check that for all $\sigma \in \texttt{createSeq}(t)$ we have the properties $\texttt{CostSeq}(\sigma) = \texttt{CostTree}(t)$, $\texttt{StateCover}(\sigma) = \texttt{StateCoverT}(t)$, and $\texttt{TranCover}(\sigma) = \texttt{TranCoverT}(t)$. We will prove these properties by induction over the structure of t. Then, each of the equalities given in (b) will be a trivial consequence of the transitivity of the equality. As anchor case, let us consider a tree $t = (s, [\,])$. For all $\sigma \in \texttt{createSeq}(t)$ we have $\sigma = s$. Thus, we have $\texttt{CostSeq}(\sigma) = \texttt{CostTree}(t) = 0$. Besides $\texttt{StateCover}(\sigma) = \texttt{StateCoverT}(t) = \{s\}$ and $\texttt{TranCover}(\sigma) = \texttt{TranCoverT}(t) = \emptyset$. Let us consider the inductive case. Let $t = (s, [(\delta_1, t_1), \ldots, (\delta_n, t_n)])$, where we consider that for all $\leq j \leq n$ we have $\delta_j = s \xrightarrow{i_j/o_j/c_j} s'_j$. If $\sigma \in \texttt{createSeq}(t)$ then σ can be split into n consecutive subsequences σ_i such that all of them but the last one finish by loading s. In particular, for all $1 \leq j \leq n-1$ we have that σ_i follows the form $\sigma_j = s \xrightarrow{i_j/o_j/c_j} \sigma'_j \xrightarrow{\psi(s)} s$ for some σ'_j, while σ_n follows the form $\sigma_n = s \xrightarrow{i_n/o_n/c_n} \sigma'_n$ for some σ'_n. Let us note that for all σ'_i we have $\sigma'_i \in \texttt{createSeq}(t_i)$. Thus, by induction hypothesis we can assume that, for all $1 \leq i \leq n$, $\texttt{CostSeq}(\sigma_i) = \texttt{CostTree}(t_i)$, $\texttt{StateCover}(\sigma_i) = \texttt{StateCoverT}(t_i)$, and $\texttt{TranCover}(\sigma_i) = \texttt{TranCoverT}(t_i)$. Now we can see that $\texttt{CostSeq}(\sigma) = \texttt{CostTree}(t) = \delta_1 + \cdots + \delta_n + \texttt{CostTree}(t_1) + \cdots + \texttt{CostTree}(t_n) + (n-1) \cdot C$. We also have $\texttt{StateCover}(\sigma) = \texttt{StateCoverT}(t) = \{s\} \cup \bigcup_{i=1}^{n} \texttt{StateCoverT}(t_i)$, as well as $\texttt{TranCover}(\sigma) = \texttt{TranCoverT}(t) = \{\delta_1, \ldots, \delta_n\} \cup \bigcup_{i=1}^{n} \texttt{TranCoverT}(t_i)$. Thus, all required equalities hold.

A.4. Proof of Proposition 5.1

Let us consider (a). If no state or transition appears more than once in t then we can trivially construct a tree $t' \in \texttt{Trees}(W')$ representing the same traversal as t and having the same cost: for each transition $(s, s', i, o, c) \in \Delta$ traversed in t, we traverse transitions $(s, \delta, i, o, c/2)$ and $(\delta, s', i, o, c/2)$ in t'. Let us suppose that some states and transitions are repeated in t. We have the following cases:

(1) There exists a sequence σ traversing a branch of t such that all states and transitions traversed by σ are traversed somewhere else in t, and σ finishes at a leaf of t. Let σ' be the suffix of σ consisting of all steps in σ from the last bifurcation node of t traversed by σ to the leaf. The sequence σ' is *useless* for covering new states and transitions, so we can remove σ' from t. All other branches departing from the same bifurcation point and leading to leaves *only* through repeated sequences can be removed too. Moreover, if *all* branches of the bifurcation point are removed then the bifurcation point becomes a leaf itself, and we have case 1 again for the part of σ that came before σ'. Otherwise, the remaining part of σ must fit into case 2.

(2) No sequence of repeated states and transitions finishes at a leaf of t. This means that all sequence σ traversing repeated states and transitions reaches some fresh state s. Let δ be the transition taken in the last step of σ, that is, the one taken to reach s. This transition must be fresh too (otherwise, s would not be fresh). The tree t' can *bypass* σ by using a direct transition from the point where σ starts to the node representing δ in W'. The cost to this direct transition is equal to or lower than the cost to taking σ. This is so because the cost of a direct transition is the cost of the shortest path connecting the departure and the destination nodes of the direct transition.

After all useless parts of t are removed as described in case 1, we construct t' from t by treating sequences traversing only fresh states/transitions as described at the beginning of this proof, and repeated parts as described in case 2. The resulting tree t' covers the same states and transitions as t, but its cost is lower than or equal to the cost of t. Besides, no state or transition of W' is repeated in t'. Thus, t' fulfills the given requirements.

Let us consider (b). Let $t' \in \texttt{Trees}(W')$. We construct $t \in \texttt{Trees}(W)$ from t' by traversing the same standard (i.e. non-direct) transitions, and for all direct transition δ traversed in t', in t we traverse the shortest path connecting both sides of the direct transition, (standard) transition by (standard) transition. The resulting tree t covers the same states and transitions as t' and has the same cost. Thus, t fulfills the proposed requirements.

A.5. Proof of Theorem 6.1

We will use similar arguments as in the proof of Theorem 3.1. Given a set of plain sequences α, we can easily compute $\texttt{ResetsIncludedCost}(\alpha, r)$, as well as $\bigcup_{\sigma\in\alpha} \texttt{StateCover}(\sigma)$ and $\bigcup_{\sigma\in\alpha} \texttt{TranCover}(\sigma)$, and compare them with K, S', Δ' in polynomial time with respect to W, S', Δ', and K, so we have $\texttt{MLS} \in \texttt{NP}$. In order to prove $\texttt{MRP} \in \texttt{NP-complete}$, we polynomially reduce the NP-complete *Hamiltonian Path problem* in directed graphs (this problem was stated before in the proof of Theorem 3.1) to MRP. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (where $\mathcal{V}$ and $\mathcal{E}$ denote the sets of vertices and edges, respectively, and we suppose $n =\mid \mathcal{V} \mid$) represent an instance of the Hamiltonian Path problem. We construct n instances of MRP from $\mathcal{G}$ as follows. Let $W_1, \ldots, W_n$ be WFSMs where, for all $1 \leq i \leq n$, we have $W_i = (S, s_i, I, O, C, \Delta)$. Note that, for all W_i, we have the same values for all tuple components but the initial state, s_i. We assume that the sets of states and transitions of all W_i, S and Δ, are defined in such a way that they literally copy the form defined by $\mathcal{V}$ and $\mathcal{E}$ in $\mathcal{G}$. The inputs and outputs labeling transitions of Δ in all WFSMs are defined in such a way that the resulting WFSM is deterministic: we consider $I = \{i_1, \ldots, i_n\}$ for some input symbols $i_1, \ldots, i_n$, $O = \{o_1, \ldots, i_n\}$ for some output symbols $o_1, \ldots, o_n$, and, for all $s \in S$, all transitions leaving s are labeled by a *different* input/output pair i_j/o_j. The cost of all transition $\delta \in \Delta$ is set to 1, and the load cost C is set to any arbitrary value. Let us note that this value is ignored to calculate the resets-included cost of a set of plain sequences, so C is irrelevant for MRP. For each W_i, we assume that its initial state s_i is set to a different state in S (thus, for all $s \in S$ there is some $1 \leq j \leq n$ such that the initial state of W_j is s). In addition, let $S' \subseteq S$ and $\Delta' \subseteq \Delta$ be defined as $S' = S$ and $\Delta' = \emptyset$, and let $K = n - 1$. Finally, let $r \in \mathbb{N}$ be equal to n. Since each W_i can be constructed from $\mathcal{G}$ in polynomial time, constructing the MRP problem instances $(W_1, S', \Delta', r, K), \ldots, (W_n, S', \Delta', r, K)$ from $\mathcal{G}$ requires polynomial time.

Let us see that the answer of the Hamiltonian Path problem for $\mathcal{G}$ is yes iff there exists $1 \leq i \leq n$ such that the answer of MRP to W_i, S', Δ', r, K is yes. Let us consider the implication from left to right. Let σ be a Hamiltonian path for $\mathcal{G}$ and W_i be the WFSM such that its initial state is the initial state of σ. We have that σ is a plain sequence for W_i, σ covers all states and transitions in S' and Δ', and $\texttt{ResetsIncludedCost}(\{\sigma\}, r) = n - 1 = K$ (note that, if our set of plain sequences contains a single sequence, then no reset cost is added). Now, let us consider the implication from right to left. If (a) α is a set of plain sequences for some W_i; (b) sequences belonging to α cover all states and transitions in S' and Δ'; and (c) $\texttt{ResetsIncludedCost}(\alpha, r) \leq K = n-1$, then there is single plain sequence in α, that is, there is no need to make any reset operation to execute all sequences in α (otherwise we would have $\texttt{ResetsIncludedCost}(\alpha, r) > K$ because we have $r = n$). Let σ be the single plain sequence belonging to α. We have $\texttt{CostPlain}(\sigma) \leq n - 1$. Since all n states of S' are covered by σ, σ must take at least $n - 1$ transitions. Note that the cost of all transitions is 1. Thus, $\texttt{CostPlain}(\sigma) \leq n - 1$ implies that σ traverses exactly $n - 1$ transitions. If σ traverses n different states in $n - 1$ transitions then no state is traversed more than once in σ. Thus, σ is a Hamiltonian path for $\mathcal{G}$.

References

[AC07] Alba E, Chicano JF (2007) Ant colony optimization for model checking. In: EUROCAST'07. LNCS, vol 4739, pp 523–530

[ADLU88] Aho A, Dahbura A, Lee D, Uyar MÜ (1988) An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman tours. In: Protocol specification, testing and verification, PSTV'88. North-Holland, Amsterdam, pp 75–86

[APH04] Antoniol G, Di Penta M, Harman M (2004) A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty. In: IEEE software metrics symposium, METRICS'04. IEEE Computer Society, New York, pp 172–183

[AT96] Alba E, Troya JM (1996) Genetic algorithms for protocol validation. In: Parallel problem solving from nature, PPSN'96. LNCS, vol 1141. Springer, Berlin, pp 870–879

[BGM91] Bernot G, Gaudel M-C, Marre B (1991) Software testing based on formal specification: a theory and a tool. Softw Eng J 6:387–405

[BL05] Burgess CJ, Lefley M (2005) Can Genetic Programming improve Software Effort Estimation? A Comparative Evaluation. In: Machine learning applications in software engineering: series on software engineering and knowledge engineering, vol 16. World Scientific Publishing Co., Singapore, pp 95–105

[Bot02] Bottaci L (2002) Instrumenting programs with flag variables for test data search by genetic algorithms. In: Conference on genetic and evolutionary computation, GECCO'02. Morgan Kaufmann Publishers Inc., Menlo Park, pp 1337–1342

[BT01] Brinksma E, Tretmans J (2001) Testing transition systems: an annotated bibliography. In: 4th summer school on modeling and verification of parallel processes, MOVEP'00. LNCS, vol 2067. Springer, Berlin, pp 187–195

[Cho78] Chow TS (1978) Testing software design modeled by finite-state machines. IEEE Trans Softw Eng 4:178–187

[CJH06] Chen K, Jiang F, Huang C-D (2006) A new method of generating synchronizable test sequences that detect output-shifting faults based on multiple UIO sequences. In: Proceedings of the 2006 ACM symposium on applied computing, SAC'06. ACM, New York, pp 1791–1797

[CKSa06] Cohen M, Kooi SB, Srisa-an W (2006) Clustering the heap in multi-threaded applications for improved garbage collection. In: Conference on genetic and evolutionary computation, GECCO'06. ACM, New York, pp 1901–1908
[Dan63] Dantzig GB (1963) Linear programming and extensions. Rand corporation research study. Princeton University Press, NJ
[DG03] Doerner K, Gutjahr WJ (2003) Extracting test sequences from a Markov software usage model by ACO. In: Conference on genetic and evolutionary computation part II, GECCO'03. LNCS, vol 2724. Springer, Berlin, pp 2465–2476
[DHHG06] Derderian K, Hierons RM, Harman M, Guo Q (2006) Automated unique input output sequence generation for conformance testing of FSMs. Comput J 49(3):331–344
[Dor04] Dorigo M (2004) Ant colony optimization. MIT Press, Cambridge
[DU95] Dahbura A, Uyar MÜ (1995) Optimal test sequence generation for protocols: the Chinese Postman algorithm applied to Q.931. in: Conformance testing methodologies and architectures for OSI protocols. IEEE Computer Society Press, New York, pp 347–351
[ELLL01] Edelkamp S, Lluch-Lafuente A, Leue S (2001) Directed explicit model checking with HSF-SPIN. In: SPIN workshop on model checking of software, SPIN'01. LNCS, vol 2057. Springer, Berlin, pp 57–79
[ENDK02] En-Nouaary A, Dssouli R, Khendek F (2002) Timed wp-method: testing real-time systems. IEEE Trans Softw Eng 28:1023–1038
[Gau95] Gaudel M-C (1995) Testing can be formal, too. In: 6th CAAP/FASE, theory and practice of software development, TAPSOFT'95, LNCS, vol 915. Springer, Berlin, pp 82–96
[GK02] Godefroid P, Khurshid S (2002) Exploring very large state spaces using genetic algorithms. In: Conference on tools and algorithms for the construction and analysis of systems, TACAS'02. LNCS, vol 2280. Springer, Berlin, pp 266–280
[Gon70] Gonenc G (1970) A method for the design of fault detection experiments. IEEE Trans Comput 19:551–558
[Har07] Harman M (2007) The current state and future of search based software engineering. In: Workshop on the future of software engineering, FOSE'07, pp 342–357
[Hen64] Hennie FC (1964) Fault detecting experiments for sequential circuits. In: Annual IEEE symposium on foundations of computer science, pp 95–110
[HHP02] Harman M, Hierons RM, Proctor M (2002) A new representation and crossover operator for search-based optimization of software modularization. In: Conference on genetic and evolutionary computation, GECC0'02. Morgan Kaufmann Publishers, Menlo Park, pp 1351–1358 Inc
[Hie02] Hierons RM (2002) Comparing test sets and criteria in the presence of test hypotheses and fault domains. ACM Trans Softw Eng Methodol 11(4):427–448
[Hie04] Hierons RM (2004) Testing from a nondeterministic finite state machine using adaptive state counting. IEEE Trans Comput 53:1330–1342
[Hie09] Hierons RM (2009) Verdict functions in testing with a fault domain or test hypotheses. ACM Trans Softw Eng Methodol 18(4):1–19
[HM09] Harman M, McMinn P (2009) A theoretical and empirical study of search-based testing: local, global, and hybrid search. IEEE Trans Softw Eng 99(RapidPosts):226–247
[Hol75] Holland JH (1975) Adaptation in natural and artificial systems. MIT Press, Cambridge
[HU10] Hierons RM, Ural H (2010) Generating a checking sequence with a minimum number of reset transitions. Autom Softw Eng 17:217–250
[HW05] Huang G-D, Wang F (2005) Automatic test case generation with region-related coverage annotations for real-time systems. In: Symposium on automated technology for verification and analysis, ATVA'05. LNCS, vol 3707. Springer, Berlin, pp 144–158
[IB07] Ipate F, Banica L (2007) W-method for hierarchical and communicating finite state machines. In: IEEE international conference on industrial informatics, vol 2. IEEE Computer Society, New York, pp 891–896
[Jon06] De Jong KA (2006) Evolutionary computation: a unified approach. MIT Press, Cambridge
[Kor92] Korel B (1992) Dynamic method for software test data generation. Softw Test Verif Reliab 2:203–213
[LHJ09] Langdon WB, Harman M, Jia Y (2009) Multi-objective higher order mutation testing with genetic programming. In: Testing: academic and industrial conference—practice and research techniques, TAIC-PART'09. IEEE Computer Society, New York, pp 21–29
[LJL07] Lam CP, Xiao J, Li H (2007) Ant colony optimisation for generation of conformance testing sequences using a characterising set. In: IASTED conference on advances in computer science and technology. ACTA Press, Mexico, pp 140–146
[LvBP94] Luo G, von Bochmann G, Petrenko A (1994) Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. IEEE Trans Softw Eng 20:149–162
[LW66] Lawler EL, Wood DE (1966) Branch-and-bound methods: a survey. Oper Res 14(4):699–719
[LY94] Lee D, Yannakakis M (1994) Testing finite-state machines: state identification and verification. IEEE Trans Comput 43:306–320
[LY96] Lee D, Yannakakis M (1996) Principles and methods of testing finite state machines: a survey. Proc IEEE 84(8):1090–1123
[LY08] Lehre PF, Yao X (2008) Crossover can be constructive when computing unique input output sequences. In: International conference on simulated evolution and learning, SEAL'08. LNCS, vol 5361. Springer, Berlin, pp 595–604
[MA01] Miller RE, Arisha KA (2001) Fault coverage in networks by passive testing. In: International conference on internet computing, IC'2001. CSREA Press, USA, pp 413–419
[McM04] McMinn P (2004) Search-based software test data generation: a survey. Softw Test Verif Reliab 14(2):105–156
[MMS01] Michael CC, McGraw G, Schatz MA (2001) Generating software test data by evolution. IEEE Trans Softw Eng 27:1085–1110
[MRR+53] Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953) Equation of state calculations by fast computing machines. J Chem Phys 21(6):1087–1092
[Pet01] Petrenko A (2001) Fault model-driven test derivation from finite state models: Annotated bibliography. In: 4th summer school on modeling and verification of parallel processes, MOVEP'00. LNCS, vol 2067. Springer, Berlin, pp 196–205
[PHP99] Pargas RP, Harrold MJ, Peck RR (1999) Test-data generation using genetic algorithms. Softw Test Verif Reliab 9:263–282

[PYB96] Petrenko A, Yevtushenko N, von Bochmann G (1996) Fault models for testing in context. In: Formal description techniques for distributed systems and communication protocols (IX), and protocol specification, testing, and verification (XVI). Chapman & Hall, London, pp 163–178

[RMN08] Rodríguez I, Merayo MG, Núñez M (2008) $\mathcal{HOTL}$: hypotheses and observations testing logic. J Log Algebr Program 74(2):57–93

[Rod09] Rodríguez I (2009) A general testability theory. In: International conference on concurrency theory, CONCUR'09. LNCS, vol 5710. Springer, Berlin, pp 572–586

[RRR07] Rabanal P, Rodríguez I, Rubio F (2007) Using river formation dynamics to design heuristic algorithms. In: Unconventional computation, UC'07. LNCS, vol 4618. Springer, Berlin, pp 163–177

[RRR08] Rabanal P, Rodríguez I, Rubio F (2008) Finding minimum spanning/distances trees by using river formation dynamics. In: Ant colony optimization and swarm intelligence, ANTS'08. LNCS, vol 5217. Springer, Berlin, pp 60–71

[RRR09a] Rabanal P, Rodríguez I, Rubio F (2009) Applying river formation dynamics to solve NP-complete problems. In: Chiong R (ed) Nature-inspired algorithms for optimisation. Studies in computational intelligence, vol 193. Springer, Berlin, pp 333–368

[RRR09b] Rabanal P, Rodríguez I, Rubio F (2009) A formal approach to heuristically test restorable systems. In: International colloquium on theoretical aspects of computing, ICTAC'09. LNCS, vol 5684. Springer, Berlin, pp 292–306

[SD85] Sabnani K, Dahbura A (1985) A new technique for generating protocol test. SIGCOMM Comput Commun Rev 15:36–43

[SD88] Sabnani K, Dahbura A (1988) A protocol test generation procedure. Comput Netw ISDN Syst 15:285–297

[SL88] Sidhu D, Leung T-K (1988) Fault coverage of protocol test methods. In: Networks: evolution or revolution. Seventh annual joint conference of the IEEE computer and communications societies, INFOCOM '88. IEEE Computer Society, New York, pp 80–85

[SL89] Sidhu DP, Leung T (1989) Formal methods for protocol testing: a detailed study. IEEE Trans Softw Eng 15(4):413–426

[SL96] Shen Y, Lombardi F (1996) Graph algorithms for conformance testing using the Rural Chinese Postman tour. SIAM J Discrete Math 9:511–528

[SR09] Srivastava PR, Rai VK (2009) An ant colony optimization approach to test sequence generation for control flow based software testing. In: Information systems, technology and management. Communications in computer and information science, vol 31. Springer, Berlin, pp 345–346

[Tas02] Tassey G (2002) The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology

[WH87] Wang B, Hutchison D (1987) Protocol testing techniques. Comput Commun 10(2):79–87

[WSJE97] Wegener J, Sthamer H, Jones BF, Eyres DE (1997) Testing real-time systems using genetic algorithms. Softw Qual Control 6:127–135

[ZHM97] Zhu H, Hall PAV, May JHR (1997) Software unit test coverage and adequacy. ACM Comput Surv 29(4):366–427

Received 18 May 2010

Accepted in revised form 11 September 2011 by Jim Woodcock

Published online 19 January 2012