Testing restorable systems by using RFD: Extended version *

Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio

Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid, 28040 Madrid, Spain
e-mail:prabanal@fdi.ucm.es, isrodrig@sip.ucm.es

Abstract. Given a finite state machine denoting the *specification* of a system, finding some short interaction sequences capable to reach 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 for 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*. Besides, some experimental results are reported.

1 Introduction

Testing constructed systems is an industrial necessity in any IT developing process. In particular, *formal testing techniques* [7, 9, 2, 13] allow 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. 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* [15, 8, 5] criteria are pursued. For instance, given a specification represented by a *finite state machine* (FSM), we could be interested in applying a test suite that allows to reach, in the IUT, all transitions or all nodes defined in the specification (assuming that the IUT were actually defined as the specification). Note that the IUT could differ from the specification. In particular, reaching a specification configuration by following a given path does not imply that the same configuration is reached in the IUT. Therefore, a test suite that allows to reach each specification state or transition by following 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. Thus, incomplete test suites can be very useful in practice.

Several graph theory techniques have been applied to find short test sequences allowing to reach, at least once, all states or transitions in an FSM specification [1, 3, 14]. In this paper we generalize testing methods allowing to reach

* Research partially supported by the Spanish MCYT project TIN2006-15578-C02-01, and the Junta de Castilla-La Mancha project PAC06-0008-6995.

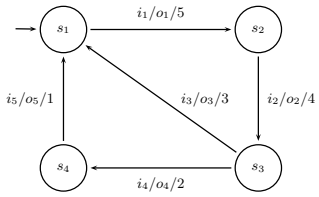

Fig. 1. Example of system

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, by using this feature, some time assigned to testing activities could be saved. Let us note that saving/restoring the complete configuration of a system could be time-expensive. In particular, 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. For instance, let us consider the (cost-weighted) FSM depicted in Figure 1. The initial state is s_1 and transitions are labelled 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 7 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 when 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).

In this paper we present a methodology that, given (i) the cost of saving/restoring a configuration; (ii) an FSM specification explicitly denoting the 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 to reach all critical configurations in a low overall time. As we will show, obtaining the *optimal* interaction sequence is an NP-hard problem. Thus, we will provide a heuristic method to approximate the optimal path. Let us note that an interaction plan where saving/restoring configurations 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 all path σ, there exists an equivalent path that can be represented by a *tree*, which implies that all loads of σ can be *sorted* in such a way that each load refers only to states appearing in the *same* branch of some tree. Thus, interaction plans found by our method will not be represented by a sequential path, but by a *tree* covering all critical points. Hence, 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 [6].In particular, we consider *River Formation Dynamics* (RFD) [10–12]. 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 on these values themselves, RFD can be seen as a *gradient-oriented* version of *Ant Colony Optimization* (ACO) [4]. However, there are other important differences between RFD and ACO (see [11, 12] for details). In particular, gradients implicitly avoid that drops get confused following local cycles (it is impossible that a path from A to A is ever-decreasing), shortcuts are quickly reinforced because they connect the same origin/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 focalized 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. Next, we formally describe the problem to be solved. In Section 3 we introduce our general heuristic method, while in Section 4 we show how to apply it to our particular problem. Afterwards, in Section 5 we report the results obtained in some experiments. Finally, in Section 6 we present our conclusions and some lines of future work.

2 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 i-th 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 1. A *Weighted Finite State Machine* (from now on `WFSM`) is a tuple (S, s_{in}, I, O, C, T) 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.
- $C \in \mathbf{N}$ is the *cost of restoring* a previously traversed state.
- $T \subseteq S \times S \times I \times O \times \mathbf{N}$ is the set of transitions. A transition $t \in T$ 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 a shorthand of $(s_1, s_2, i, o, c) \in T$.

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

$$\| \{ s \xrightarrow{i/o/c} s' \mid \exists\ o, c, s' : s \xrightarrow{i/o/c} s' \in T \} \| \leq 1$$ □

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

Activities of `WFSMs` will be denoted by *load sequences*. Each step in a load sequence consists in either taking some `WFSM` transition 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 cover some given states and/or transitions. Thus, constraining the notion of load sequence in such a way that some *useless* sequences are discarded from scratch will allow us to reduce the searching space. 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. Sequences (regardless of whether they fulfill the additional condition or not) and sequences that *actually* fulfill it will be called *α-load sequences* and *load sequences*, respectively. Later we will show that constraining our search to sequences fulfilling the additional condition does not limit the possibility to find cheap sequences.

Definition 2. An *α-load sequence* is a sequence $\sigma = s_1 \xrightarrow{t_1} s_2 \ldots s_{n-1} \xrightarrow{t_{n-1}} s_n$ where, for each $1 \leq k \leq n-1$, t_k is either $i_k/o_k/c_k$ (if $s_k \xrightarrow{i_k/o_k/c_k} s_{k+1} \in T$) or $\psi(s_j)$ (if $s_j \in \{s_1, ..., s_k\}$ and $s_{k+1} = s_j$).

In addition, σ is also a *load sequence* if for all $s, s' \in S$ such that the first appearance of s 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 $l < j$) there do not exist t_p and t_q with $p < q$ such that $t_p = \psi(s)$ and $t_q = \psi(s')$.

The set of all α-load sequences of a `WFSM` W is denoted by α-`Sequences`(W). The set of all load sequences of W is denoted by `Sequences`(W).

Given $\sigma = s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-2}} s_{n-1} \xrightarrow{t_{n-1}} s_n \in \alpha$-`Sequences`$(W)$, we consider $\sigma^- = s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-2}} s_{n-1}$.

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

Clearly, α-$\texttt{Sequences}(W) \subset \texttt{Sequences}(W)$. For instance, let us consider $\sigma = s_1 \xrightarrow{i_1/o_1/c_1} s_2 \xrightarrow{i_2/o_2/c_2} s_3 \xrightarrow{\psi(s_1)} s_1 \xrightarrow{i_3/o_3/c_3} s_4 \xrightarrow{\psi(s_2)} s_2 \xrightarrow{i_4/o_4/c_4} s_5$. We have $\sigma \in \alpha$-$\texttt{Sequences}(W)$ for some $\texttt{WFSM}$ W, but $\sigma \notin \texttt{Sequences}(W)$ (see states s_1 and s_2 and the positions of $\psi(s_1)$ and $\psi(s_2)$ in σ). Let us consider $\sigma' = s_1 \xrightarrow{i_5/o_5/c_5} s_6 \xrightarrow{i_7/o_7/c_7} s_7 \xrightarrow{i_8/o_8/c_8} s_8 \xrightarrow{\psi(s_7)} s_7 \xrightarrow{i_9/o_9/c_9} s_{10} \xrightarrow{\psi(s_6)} s_6$. We have $\sigma' \in \texttt{Sequences}(W)$ (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. 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 a making a *load*.

Definition 3. Let $\sigma = s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n \in \alpha$-$\texttt{Sequences}(W)$. If $1 \leq k \leq n-1$ then the *cost* of t_k, denoted by $\texttt{CostTran}(t_k)$, is defined as

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

The *cost* of σ, denoted by $\texttt{CostSeq}(\sigma)$, is defined as $\sum_{k=1}^{n-1} \texttt{CostTran}(t_k)$. □

Given the α-load sequence σ considered before, we have $\texttt{CostSeq}(\sigma) = c_1 + c_2 + C + c_3 + C + c_4$. Next we show that, by constraining our search to load sequences (instead of considering all α-load sequences) we do not lose the possibility to find sequences whose cost is under some given upper bound.

Lemma 1. For all $\sigma \in \alpha$-$\texttt{Sequences}(W)$ there exists $\sigma' \in \texttt{Sequences}(W)$ such that $\texttt{CostSeq}(\sigma) = \texttt{CostSeq}(\sigma')$.

Proof. Let $\sigma = \sigma_1 \cdot \ldots \cdot \sigma_n$ be such that for all $1 \leq j \leq n$ we have $\sigma_j = s_{1,j} \xrightarrow{t_{1,j}} \ldots \xrightarrow{t_{k_j,j}} s_{k_j,j}$, where $t_{k_j,j} = \psi(s_{k_j,j})$ and for all $1 \leq l < k_j$ we have $t_{l,j} = i_{l,j}/o_{l,j}/c_{l,j}$ 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:

$R := \{(s_{1,j}, (\sigma_j)^-) | 2 \leq j \leq n\}$;
$\sigma' := (\sigma_1)^-$;
$stack :=$ a stack with a single member $s_{1,1}$;
while $R \neq \emptyset$ **do**
  $first := top(stack)$;
  while $\nexists\, \sigma''((first, \sigma'') \in R)$ **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 in $stack$ 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 $\sigma' \in \texttt{Sequences}(W)$ and $\texttt{CostSeq}(\sigma) = \texttt{CostSeq}(\sigma')$. 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 σ'' such that $(first, \sigma'') \in R$. By contradiction, let us suppose that *stack* is fully emptied without finding such σ''. This means that no element $(s, \sigma'') \in R$ is such that $s \in stack$. For all $(s, \sigma'') \in R$, either s has already been traversed by σ' or not. If it does then, since all states traversed by σ' are eventually introduced in *stack* but we have $s \notin stack$, then 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 $(s, \sigma'') \in R$, s has not been traversed yet by σ'. However, since σ is the result of connecting all σ'' such that $(s, \sigma'') \in R$ with appropriate loads, and loads cannot lead to *new* states, for all σ'' included so far in σ' there must exist $(s, \sigma''') \in R$ such that s appears in σ''. Therefore, for some $(s, \sigma'') \in R$ we have that s has been traversed by σ' and we have a contradiction.

Next we consider (b). Let $\sigma' = s_1 \xrightarrow{t_1} s_2 \dots s_{n-1} \xrightarrow{t_{n-1}} s_m$ be the sequence returned after terminating the algorithm. New 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 $1 \leq i < j \leq m$ 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 $t_p = \psi(s_i)$ and $t_q = \psi(s_j)$. Hence, $\sigma' \in \texttt{Sequences}(W)$. Besides, let us note that σ' includes the same transitions of σ, as well as the same number of loads. Thus, $\texttt{CostSeq}(\sigma) = \texttt{CostSeq}(\sigma')$. □

Let us revisit σ. Since $\sigma \in \alpha\text{-}\texttt{Sequences}(W)$, we have $\sigma'' \in \texttt{Sequences}(W)$ for $\sigma'' = s_1 \xrightarrow{i_1/o_1/c_1} s_2 \xrightarrow{i_2/o_2/c_2} s_3 \xrightarrow{\psi(s_2)} s_2 \xrightarrow{i_4/o_4/c_4} s_5 \xrightarrow{\psi(s_1)} s_1 \xrightarrow{i_3/o_3/c_3} s_4$. The property $\texttt{CostSeq}(\sigma) = \texttt{CostSeq}(\sigma'')$ trivially holds because we traverse the same transitions and we make the same number of loads.

The condition imposed to α-load sequences to be considered as *load sequences* implies that, in particular, load sequences can be equivalently represented as *trees*. A *load tree* is a tree-like version of a load sequence, where loads are represented by *bifurcations* in the tree. In particular, a bifurcation represents that, after we complete one of the choices appearing at the bifurcation (i.e. after we complete the *subtree* representing this choice), we will restore the bifurcation state 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.

Definition 4. A *load tree* is a term t belonging to the language induced by the term Lt in the following E-BNF:

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

such that, if t follows the form $t = (st, [(tr_1, child_1), ..., (tr_n, child_n)])$, then for all $1 \leq k \leq n$ the conditions $child_k \neq empty$ and $st \xrightarrow{tr_k} \texttt{fst}(child_k) \in T$ must hold.

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 5. Given a tree $t = (root, children)$, the cost of t, denoted by $\texttt{CostTree}(t)$, is defined as

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

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 *preorder* 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 returned trees are taken as *children* of t. In functions `createSeq` and `createSeq` given in the next definition, anchor cases are again reached when the target tree has no children. Besides, note that, 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 6. Let $t_1 = (root_1, children_1)$ and $t_2 = (root_1, children_2)$ 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

- $root_1 = root_2$,
- $n_1 = n_2$,
- 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 $\texttt{snd}(children_1[i]) \equiv_T \texttt{snd}(children_2[f(i)])$ (i.e. subtrees are equivalent).

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

$$\texttt{createTree}(\sigma) = (s_1, [(t_{1,1,}, \texttt{createTree}(\sigma_1)), ..., (t_{1,n}, \texttt{createTree}(\sigma_n))])$$

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

The *set of sequences of a tree* $t = (root, children)$, denoted by $\texttt{createSeq}(t)$, is defined as

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

Lemma 2. We have the following properties:

- $\texttt{createSeq}(t) = \{\sigma | \sigma \rhd t\}$
- for all $\sigma, \sigma' \in \texttt{createSeq}(t)$ we have $\texttt{CostSeq}(\sigma) = \texttt{CostSeq}(\sigma') = \texttt{CostTree}(t)$.

□

The previous result guarantees that, if we want to find the cheapest load sequence for a given WFSM, then we can concentrate on searching the equivalent cheapest load tree. Given such cheapest tree t, the load sequence to be used can be any sequence σ such that $\sigma \in \texttt{createSeq}(t)$.

Next we define the target problem of this paper.

Definition 7. Given a WFSM $W = (S, s_0, I, O, C, T)$, some sets $S' \subseteq S$ and $T' \subseteq T$, 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 $\sigma \in \texttt{Sequences}(W)$ such that $\texttt{CostSeq}(\sigma) \leq K$ and, for all $s \in S'$ and $t \in T'$, s' and t' appear in σ? □

As far as we are concerned, MLS has not been studied before in the literature. Thus, before describing our heuristic method to solve it, we prove that it is an NP-complete problem.

Theorem 1. $\texttt{MLS} \in \texttt{NP-complete}$.

Proof. Given a sequence σ, we can easily compute $\texttt{CostSeq}(\sigma)$ (and compare it with K) in polynomial time with respect to W, S', T', and K, so we have $\texttt{MLS} \in \texttt{NP}$. In order to prove $\texttt{MLS} \in \texttt{NP-complete}$, 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 to 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 vertexes and edges, respectively, and we suppose $n = |\mathcal{V}|$) represent 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 $W_i = (S, s_i, I, O, C, T)$ (note that, for all W_i, we have the same values for all tuple components but the initial state, s_i). Sets S and T, 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 T 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 $t \in T$ 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 $T' \subseteq T$ be defined as $S' = S$ and $T' = \emptyset$, 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', T', K), ..., (W_n, S', T', 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', T', 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 T', 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 T'; 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 $\mathcal{G}$. □

3 Brief Introduction to River Formation Dynamics

In this section we briefly introduce the basic structure of River Formation Dynamics (RFD) (for further details, see [10, 12]). 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 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 [10, 12]).

Compared to a related well-known evolutionary computation method, *Ant Colony Optimization*, 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 to be led by pheromone trails through some edges in such a way that a node must be repeated in the next step[1]. 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 (in 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.

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

4 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 make this node be 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, edges 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 *uncomplete* covering, drops following this path 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, 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 strongly 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 construction 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 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 not critical) nodes and transitions $N_1 \rightarrow \ldots \rightarrow N_m$ 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 $A \rightarrow N_1 \rightarrow \ldots \rightarrow N_m \rightarrow B$. 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 $N_1 \rightarrow \ldots \rightarrow N_m$ 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 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 be 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 using *only* this alternative notion of direct edge (that is, not considering direct edges leading to *states*) does not disable our first example in 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 $N_1 \xrightarrow{i_1/o_1/c_1} B$, ..., $N_m \xrightarrow{i_m/o_m/c_m} B$. 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 $c'_j + (c_j/2)$. Since the transition connecting this barrier node with B will have cost $c_j/2$, the total cost of moving from A to B through N_j will be $c'_j + c_j$, as expected.

In order to (implicitly) allow repetitions, adding direct transitions is, in general, a better choice than adding several instances of each node. Let us note that if we have to repeat some nodes to reach X 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 in a graph with several instances of each node is inefficient because the Floyd algorithm *optimally* solves this problem in polynomial time. Thus, adding direct transitions is a good choice unless repeating nodes is *rarely* preferable (e.g. this could be the case 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. 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 should 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 longer 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 longer.

5 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 allow to 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 being 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 a branch and bound (B&B) strategy.

All experiments were performed in an Intel T2400 processor with 1.83 Ghz. RFD was executed fifty times for each of the graphs during five minutes, while the B&B method was executed (only once) during one hour (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 (average), the variance, and the best solution found for the alternative case where we cannot load any state different to the initial state (reset).

In Table 1, symbol '–' denotes a non-applicable case. The input 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 (x), each node is connected with the 80% of the nodes. We present the results when the load cost is relatively cheap (20), when the load cost is medium (100) and when the load cost is high (1000) with respect to the cost of an edge of the `WFSM`. Furthermore, the cost associated to a reset of the system is set to 100, that is, the average cost of traversing two edges in 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 *Reset*). Let us notice 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 *Reset* are nearly equal. However, in the case of sparse graphs (i.e. in the most frequent practical case) the advantages are obvious: In nearly all the cases the column *Best* outperforms the column *Reset*. 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 remind that RFD was executed during only five minutes while the B&B strategy was running during one hour. 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.

6 Conclusions and Future Work

In this paper we have 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 possible when the IUT is a software system. Beyond this requirement, 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. In this case,

Method	Graph size	Sparse	Load cost	Best	Average	Variance	Reset
RFD	50	✓	20	1724	2611	411299	6494
B&B	50	✓	20	2036	-	-	13257
RFD	50	✓	100	2380	3499	121251	6248
B&B	50	✓	100	2895	-	-	7142
RFD	50	✓	1000	4062	4401	45120	3482
B&B	50	✓	1000	7306	-	-	8918
RFD	100	✓	20	3678	4118	51470	17569
B&B	100	✓	20	12670	-	-	14982
RFD	100	✓	100	4054	4669	276078	8775
B&B	100	✓	100	12830	-	-	14982
RFD	100	✓	1000	5429	5793	45082	5429
B&B	100	✓	1000	14630	-	-	14982
RFD	200	✓	20	5573	6756	530882	25266
B&B	200	✓	20	32819	-	-	69656
RFD	200	✓	100	7653	8260	73943	12670
B&B	200	✓	100	33219	-	-	69656
RFD	200	✓	1000	18198	19524	1168366	18198
B&B	200	✓	1000	37719	-	-	69656
RFD	50	x	20	265	288	164	296
B&B	50	x	20	566	-	-	566
RFD	50	x	100	244	270	141	244
B&B	50	x	100	580	-	-	580
RFD	50	x	1000	242	273	107	242
B&B	50	x	1000	580	-	-	580
RFD	100	x	20	246	278	616	246
B&B	100	x	20	607	-	-	607
RFD	100	x	100	245	284	1090	245
B&B	100	x	100	607	-	-	607
RFD	100	x	1000	242	262	86	242
B&B	100	x	1000	607	-	-	607
RFD	200	x	20	300	336	293	300
B&B	200	x	20	367	-	-	367
RFD	200	x	100	297	347	583	297
B&B	200	x	100	367	-	-	367
RFD	200	x	1000	277	336	685	277
B&B	200	x	1000	367	-	-	367

Table 1. Summary of results.

using load/restore operations allows to reduce 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*). Thus, in order to solve this problem, we have applied an evolutionary computation approach, 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 river formation dynamics 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` should be studied. Previous studies show that RFD fits particularly well for problems consisting in forming a kind of covering tree [11, 12], 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. In particular, RFD adaptations considered in [12] 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 actually deal with EFSMs should be studied.

References

1. A. Aho, A. Dahbura, D. Lee, and M.Ü. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman tours. In *Protocol Specification, Testing and Verification VIII*, pages 75–86. North Holland, 1988.
2. E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In *4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067*, pages 187–195. Springer, 2001.
3. A. Dahbura and M. Uyar. Optimal test sequence generation for protocols: The chinese postman algorithm applied to Q.931. In *Conformance testing methodologies and architectures for OSI protocols*, pages 347–351. IEEE Computer Society Press, 1995.
4. M. Dorigo. *Ant Colony Optimization*. MIT Press, 2004.
5. G.-D. Huang and F. Wang. Automatic test case generation with region-related coverage annotations for real-time systems. In *3rd Int. Symposium on Automated Technology for Verification and Analysis, ATVA'05, LNCS 3707*, pages 144–158. Springer, 2005.
6. K.A. De Jong. *Evolutionary computation: a unified approach*. MIT Press, 2006.
7. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
8. R.E. Miller and K.A. Arisha. Fault coverage in networks by passive testing. In *International Conference on Internet Computing 2001, IC'2001*, pages 413–419. CSREA Press, 2001.
9. A. Petrenko. 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 2067*, pages 196–205. Springer, 2001.
10. P. Rabanal, I. Rodríguez, and F. Rubio. Using river formation dynamics to design heuristic algorithms. In *Unconventional Computation, UC'07, LNCS 4618*, pages 163–177. Springer, 2007.
11. P. Rabanal, I. Rodríguez, and F. Rubio. Finding minimum spanning/distances trees by using river formation dynamics. In *Ant Colony Optimization and Swarm Intelligence, ANTS'08, LNCS 5217*. Springer, 2008.
12. P. Rabanal, I. Rodríguez, and F. Rubio. Applying river formation dynamics to solve NP-complete problems. In R. Choing, editor, *Nature-Inspired Algorithms for Optimisation*, volume 193 of *Studies in Computational Intelligence*. Springer, 2009.
13. I. Rodríguez, M.G. Merayo, and M. Núñez. $\mathcal{HOTL}$: Hypotheses and observations testing logic. *Journal of Logic and Algebraic Programming*, 74(2):57–93, 2008.
14. Y. Shen and F. Lombardi. Graph algorithms for conformance testing using the rural chinese postman tour. *SIAM Journal on Discrete Mathematics*, 9:511–528, 1996.
15. H. Zhu, P.A.V Hall, and J.H.R. May. Software unit test coverage and adequacy. *ACM Computing Surverys*, 29(4):366–427, 1997.