Synthesis of Test Purpose Directed Reactive Planning Tester for Nondeterministic Systems

Jüri Vain
Dept. of Comp. Sci.
Tallinn Univ. of Technology
Raja 15
12618 Tallinn, Estonia
vain@ioc.ee

Kullo Raiend
Elvior
Mustamäe tee 44
10621 Tallinn, Estonia
kullo.raiend@elvior.ee

Andres Kull
Elvior
Mustamäe tee 44
10621 Tallinn, Estonia
andres.kull@elvior.ee

Juhan P. Ernits
Institute of Cybernetics
Akadeemia 21
12618 Tallinn, Estonia
juhan@cc.ioc.ee

ABSTRACT
We describe a model-based construction of an online tester for black-box testing of the implementation under test (IUT). The external behaviour of the IUT is modelled as an output observable nondeterministic EFSM with the assumption that all transition paths are feasible. A test purpose is attributed to the IUT model by a set of Boolean variables called traps that are used to measure the progress of the test run. These variables are associated with the transitions of the IUT model. The situation where all traps have been reached means that the test purpose has been achieved. We present a way to construct a tester that at runtime selects a suboptimal test path from trap to trap by finding the shortest path to the next unvisited trap. The principles of reactive planning are implemented in the form of the decision rules of selecting the shortest paths at runtime. The decision rules are constructed in advance from the IUT model and the test purpose. Preliminary experimental results confirm that this method clearly outperforms random choice and is better than the anti-ant algorithm in terms of resultant test sequence length to achieve the test purpose.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging—Testing tools; I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search—Plan execution, formation, and generation

General Terms
Algorithms, Reliability

Keywords
Model-based testing, Nondeterministic extended finite state machine, Online testing, Reactive planning

1. INTRODUCTION
On-the-fly testing is widely considered to be the most appropriate technique for model-based testing of an implementation under test (IUT) modelled using nondeterministic models [18, 19]. We use the term on-the-fly to describe a test generation and execution algorithm that computes successive stimuli incrementally at runtime, directed by the test purpose and the observed outputs of the IUT.

The state-space explosion problem experienced by many offline test generation methods is reduced by the on-the-fly techniques because only a limited part of the state-space needs to be kept track of at any point in time. On the other hand, exhaustive planning is difficult on-the-fly due to the limitations of the available computational resources at the time of test execution.

The simplest approach to the selection of test stimuli is to apply the so called random walk strategy where no test sequence has an advantage over the others. It is inefficient because it is based on the random exploration of the state space and leads to test cases that are unreasonably long and nevertheless may leave the test purpose unachieved. To overcome this deficiency additional heuristics are applied for guiding the exploration of the state space [11, 20]. The other extreme of guiding is exhaustive planning by solving constraint systems at each step. For instance, the witness trace generated by model checking provides possibly optimal selection of the next test stimulus. The critical issue in the case of explicit state model checking algorithms is the size and complexity of the model leading to the explosion of the state space specially in cases such as “combination lock” or deep loops in the model [7].

In this paper we propose a balance between the trade-offs of using a simple heuristic and the exhaustive planning methods for on-the-fly testing. We apply the principles of reactive planning to the problem of test planning under uncertainty. Reactive planning operates in a timely fashion and hence can cope with highly dynamic and unpredictable...
environments [22]. Just one subsequent input is computed at every step, based on the current context. Instead of producing a complete test plan with branches (test tree), a set of decision rules is produced. We construct these rules by offline analysis based on the given IUT model and the test purpose.

The key assumption is that the IUT model is presented as an output observable nondeterministic state machine [12, 16] either in the form of an FSM or an EFSM in which all transition paths are feasible [5, 8]. From the IUT model we synthesise a reactive planning tester that is able to generate test inputs on-the-fly depending on the observed reactions of the IUT and the test purpose without having a preset test tree generated in advance. The proposed approach leads to a tester that directs the IUT efficiently towards the user-defined test purpose during test execution.

A test purpose is a specific objective or a property of the IUT that the tester is set out to test. We focus on test purposes that can be defined as a set of traps associated with the transitions of the IUT model [7]. The goal of the tester is to generate a test sequence so that all traps are visited at least once during the test.

We synthesise the tester as an EFSM where the rules for online planning derived during the tester synthesis are encoded into the transition guards of the EFSM. At each step only the rules associated with the outgoing transitions of the current state of the EFSM are evaluated to select the next transition with the highest gain. Thus, the number of rules that need to be evaluated at each step is relatively small.

The decision rules are constructed taking into account the reachability of all trap-equipped transitions from a given state and the length of the paths to them. Also, the current value (visited or not) of each trap is taken into account. The decision rules are derived by performing reachability analysis from the current state to all trap-equipped transitions by constructing the shortest path trees. The gain functions that are the terms of the decision rules are derived from the shortest path trees by simple rewrite rules.

The resulting tester drives the IUT from one state to the next by generating inputs and by observing the outputs of the IUT. When generating the next input the tester takes into account which traps have been visited in the model before. The execution of the decision rules at the time of the test execution is significantly faster than finding the efficient test path by state space exploration algorithms but nevertheless leads to the test sequence that is lengthwise close to optimal.

2. RELATED WORK

In on-the-fly testing the test generation procedure derives only one test input at a time from the model and feeds it immediately to the IUT as opposed to deriving a complete test case in advance like in offline testing. It it not required to explore the whole state space of the model of the IUT at any time, instead, the decisions about next actions are made by observing the current output of the IUT [21]. However, on-the-fly test execution requires more runtime resources for interpreting the model.

The simplest on-the-fly test input selection algorithm is random choice. Random choice has been used in early TorX tool [2], Uppaal-Tron [14, 10] and also in the on-the-fly testing mode of SpecExplorer [15]. In [6] a transition probabilities directed next input selection is introduced to TorX.

Test purpose based test selection algorithms reduce the total number of states to be explored in a model by test purposes that are formalised as observation objectives, which can be hit or missed when executing a test. The test purposes approach was formally elaborated in [4] and used in TorX [17] and TGV [9]. Our reactive planning tester uses similar test purposes to guide the planning. A further development of the SpecExplorer approach has been introduced in NModel [18, 19] where the IUT model presented as a model program can be composed with scenario models to test certain scenarios which are subsets of all possible behaviours.

An anti-ant [11] based algorithm of reinforcement learning [20] is used to cover all transitions of the labelled transition system resulting in exploring a model program. The selection of the next input from the alternatives tries to avoid taking already visited transitions. The main difference is that in our case the planning looks ahead more than one step at a time to reach still unsatisfied parts of the test purpose, but the anti-ant approach looks only one step ahead when selecting the least visited outgoing transition from a state.

The concept of a reactive planner as presented in [22] is motivated by work with model-based autonomy. The idea is that as much as possible of the combinatorially hard planning towards a specified goal is done in advance and is recorded into the rules of the planner which in turn get fired when relevant criteria are satisfied.

Our approach is similar: we present an algorithm for creating a tester that tests the IUT and terminates when a prescribed test purpose is satisfied. As the specification model may be nondeterministic, it is impossible to predict exactly how long such test should take but under the fairness assumption all choices will eventually be possible and thus the test purpose becomes fulfilled.

Additionally, the reactive planning tester is able to guide the model on-the-fly towards still unexplored areas even in cases where well explored parts of the model need to be traversed. The anti-ant algorithm strictly prefers less visited transitions to more visited ones.

3. MODEL-BASED TESTING WITH EFSMS

3.1 Extended Finite State Machine

Our approach is about synthesising the online tester model for the IUT that is modelled by a nondeterministic EFSM. The IUT model is restricted to a subclass of EFSMs where all possible sequences of transitions are feasible. In general, an EFSM model can contain infeasible sequences of transitions as the current configuration of context variables at some state may make some of the guards of the outgoing transitions from that state evaluate to false. There are algorithms described in [5, 8] for transforming an EFSM into a form where all paths are feasible.

**Definition 1:** An extended finite state machine (EFSM), $M$ is defined as a tuple $(S, \Sigma, I, O, E)$, where $S$ is a finite set of states, $\Sigma$ is a finite set of inputs, $I$ is the finite set of initial states, $O$ is the finite set of outputs, and $E$ is the set of transitions. A configuration of $M$ is a pair $(s, \sigma)$ where $s \in S$ and $\sigma \in \Sigma$ is a mapping from $V$ to values, and $V$ is a finite set of variables with finite value domains. $I$ is the finite set of inputs, $O$ is the finite set of outputs, and $E$ is the set of transitions. A configuration of $M$ is a pair $(s, \sigma)$ where $s \in S$ and $\sigma \in \Sigma$ is a mapping from $V$ to values, and $V$ is a finite set of mappings from variable names to their possible values. The initial configuration is $(s_0, \sigma_0)$, where $\sigma_0 \in \Sigma$ is the initial assignment. A transition $e \in E$ is a tuple $e =$
\((s, p, a, o, u, q)\), where \(s\) is the source state of the transition, \(q\) is the target state of the transition \((s, q \in S)\), \(p\) is a transition guard that is a logic formula over \(V\), \(a\) is the input of \(M\) \((a \in I)\), \(o\) is the output of \(M\) \((o \in O)\), and \(u\) is an update function over \(V\).

A deterministic EFSM is an EFSM where the output and next state are unambiguously determined by the current state and the input. A nondeterministic EFSM may contain states where the reaction of the EFSM in response to an input is nondeterministic i.e. there are more than one outgoing transitions that are enabled simultaneously.

### 3.2 Model of the IUT

The model of the IUT is an EFSM denoted by \(M_S\) and it can be either deterministic or nondeterministic, it can be strongly connected or not. If the model is not strongly connected then we assume that there exists a reliable reset function over \(V\).

It is essential that the tester can observe the outputs of the IUT for detecting the next state after a nondeterministic transition of the IUT. Therefore, we require that a nondeterministic IUT is output observable \([12, 16]\) which means that even though there may be multiple transitions taken in response to a given input, the output identifies the next state unambiguously.

An example of an output observable nondeterministic IUT model is given in Figure 1 a). The outgoing transitions \(e_0\) and \(e_1\) \((c_3\) and \(c_4\)\) of the state \(s_1\) \((s_2)\) have the same input \(a_0\) \((a_3)\), but different outputs \(a_0\) or \(o_1\) \((a_3\) or \(a_4)\).

![Figure 1: a) An output observable nondeterministic IUT model. b) The above model extended with trap variables.]

### 3.3 Test Purpose

A test purpose is a specific objective or a property of the IUT that the tester is set out to test. In general test purposes are selected based on the correctness criteria stipulated by the specification of the IUT. The goal of specifying test purposes is to establish some degree of confidence that the IUT conforms to the specification. In model-based testing the formal model of the IUT is derived from the specification and it is the starting point of the automatic test case generation. Therefore, it should be possible to map the test purposes derived from the specifications of the IUT into test purposes defined in terms of the IUT model. Examples of our test purposes are "test a state change from state A to state B in a model", "test whether some selected states of a model are visited", "test whether all transitions are visited at least once in a model", etc. All of the test purposes listed above are specified in terms of the structural elements of the model that should be traversed during the execution of the test. A tester model is generated from the IUT model attributed with such test purpose. The tester runs until the test purpose is achieved and guides the IUT at runtime towards still unsatisfied parts of the test purpose.

### 3.4 Encoding the Test Purpose into the IUT Model

For synthesising a tester that fulfills a particular test purpose we extend the original model of the IUT with traps and generate the tester from the extended model of the IUT. The traps are attached to the transitions of the IUT model and they can be used to define which model elements should be visited by the test.

The traps are implemented by trap variables and trap update functions. A trap variable is a Boolean variable initially set to \(false\). The trap update functions are attached to the transitions of the model and they are executed when the transition is visited during the execution of the test. The trap update functions are used to set trap variables to \(true\) which denotes visited traps.

The extended model of the IUT \(M'_S\) is a tuple \((S'_S, V'_S, I'_S, O'_S, E'_S)\). The extended set of variables \(V'_S\) includes variables of the IUT and the trap variables \((V'_S = V_S \cup T)\), where \(T\) is a set of trap variables. \(E'_S\) is a set of transitions where each element of \(E'_S\) is a tuple \(s, p', a, o, u', q\), where \(p'\) is a transition guard that is a logical formula over \(V'_S\), and \(u'\) is an update function over \(V'_S\). For the sake of brevity we further denote the model of the IUT that is extended with trap variables by \(M'_S\).

Figure 1 b) presents an example where the IUT model given in Figure 1 a) is extended with trap variables. The example presents a visit all transitions test purpose, therefore the traps are attached to all transitions, \(T = \{t_0, \ldots, t_r\}\).

In this example \(V'_S = T\) and \(p_k \equiv true, u_k \equiv \delta_k := true\) for each transition \(e_k, k \in \{0, \ldots, r\}\).

### 3.5 Model of the Tester

The tester is synthesised from the extended IUT model \(M'_S\). Its structure is derived from the structural elements of \(M'_S\) – states, transitions, variables, and update functions. We synthesise a tester EFSM \(M_T\) as a tuple \((S_T, V_T, I_T, O_T, E_T)\), where \(S_T\) is the set of tester states, \(V_T\) is the set of tester variables, \(I_T\) is the set of tester inputs, \(O_T\) is the set of tester outputs and \(E_T\) is the set of tester transitions. Running the test presumes that the tester inputs are con-
MI and MR set up the planning problem, identifying initial (MI) and model-based reactive planning (MRP) \[22\]. The tester has two types of states - active and passive. The set of active states $S_T^a$ ($S_T^a \subset S_T$) includes the states where the IUT is idle and the tester controls the test execution. The set of passive states $S_T^p$ ($S_T^p \subset S_T$) includes the states where the tester is idle and the control is in the IUT side.

The transitions $e_T \in E_T$ of the tester automaton are defined by a tuple $(s_T, p_T, q_T, o_T, e_T)$, where $p_T$ is a transition guard that is a logical formula over $V_T$ and $u_T$ is an update function over $V_T$. We distinguish observable and controllable transitions of the tester. An observable transition $e_T$ is a transition originating from a passive state of the tester. It is defined by a tuple $(s_T, p_T \equiv true, a_T, o_T \equiv nil, u_T, q_T)$, where $s_T$ is a passive state, the transition is always enabled ($p_T \equiv true$), and it does not expect any output from the tester. A controllable transition $e_T$ is a transition originating from an active state of the tester. It is defined by a tuple $(s_T, p_T, a_T, o_T, u_T \equiv nil, q_T)$, where $s_T$ is an active state, the transition does not contain an input, $p_T \equiv ps$ and $p_T(V_T)$ is a guard of $e_T$ constructed as a conjunction of the corresponding guard $ps$ of the extended IUT model $M_S$ and the guard $p_T(V_T)$. The purpose of the gain guard is to guide the tester in selecting the next transition from the set of outgoing transitions of the current state to reach the next unvisited trap.

The gain guard must ensure that in the active state of the tester only those outgoing transitions are enabled that have the maximum gain. The enabled transition is the best choice in the sense of the path length from the current state towards fulfilling a still unsatisfied subgoal of the test purpose. We construct the gain guards $p_T(V_T)$ offline using the reachability analysis of the traps from the given transition. The gain guards take into account the amount and distance-weighted reachability (gain) of still unvisited traps. The tester model can be non-deterministic in the sense that when there are many transitions with equal positive gain, the selection of the transition to be taken next is made randomly from the best choices.

4. TESTER CONSTRUCTION ALGORITHM

4.1 A Model-Based Reactive Planning Tester

We apply the concept of reactive planning to tester synthesis. Reactive planning is typically used in agents operating in uncertain and dynamic environments \[13, 22\].

The idea of a reactive executor is that it continually tries to take the system toward a state that satisfies the desired goals. It is reactive in the sense that it reacts immediately to observed outputs of the IUT and to changes in the goals. For example, in the case of testing, each input of the IUT $a_i$, where $i$ denotes each individual transition, is incrementally generated using the new information from observations and goal configurations determined by the test purpose.

A model-based executive uses a specification of a transition system to determine the desired control sequence in three stages - mode identification (MI), mode reconfiguration (MR) and model-based reactive planning (MRP) \[22\]. MI and MR set up the planning problem, identifying initial and target states, while MRP reactively generates a plan solution. MI is a phase where the current state of the EFMS is identified. In the case of a deterministic transition MI is trivial, it is just the next state corresponding to the IUT input $a_i$. In the nondeterministic case, MI can determine the current state by looking at the output $o_i$ due to the output observability assumption. In the current approach the MR and the MRP phases are combined into one since both the goal and the next step toward the goal are determined by the same decision tree as explained later.

**Definition 2:** A model-based reactive planner, MRP, (a modification of Def. 2 in \[22\]) takes as input a specification of a transition system $M_S$, a current source state $s_i$ (from MI), and the lowest cost next transition $e_i$ that takes us closer to the still unsatisfied set of subgoals $\{t_i = false\}$ (from MR). By taking $e_i$, the system arrives in the target state $q_i$. The MRP generates an IUT input $a_i$ such that for any assignment $\sigma_i \in \Sigma$ that agrees with $s_i$ and $a_i$, the state $s_{i+1}$ either satisfies one of the previously unsatisfied subgoals or enables the IUT to come one step closer to satisfying one.

4.2 Control Structure of the Tester

The tester model is constructed as a dual automaton of the IUT model where the inputs and outputs are inverted. The tester construction algorithm, Algorithm 1, has the following steps. The states of the IUT model are transformed into the active states of the tester model in step 1. For each state $s$ of the IUT, the set of transitioning $E_{S_{out}}^o(s)$ in steps 2 to 5. The transitions of the IUT model are split into two transitions of the tester model - controllable transition $e_T \in E_T$ and observable transition $e_T \in E_T$ where $E_T$ and $E_T$ are subsets of controllable and observable transitions of the tester. A new intermediate passive state $s_T$ is added between them (steps 6 – 8).

Let $E_{S_{out}}^o(s,a,p)$ denote a subset of the nondeterministic outgoing transitions of the state $s$ where the IUT input is $a$ and the guard is equivalent to $p$. The algorithm creates a controllable transition $e_T$ for each set $E_{S_{out}}^o(s,a,p)$ from state $s$ to the passive state $s_T$ of the tester model (step 7). The controllable transition $e_T$ does not have any input and the input of the corresponding transition of the IUT becomes an output of $e_T$.

For each element $e \in E_{S_{out}}^o(s,a,p)$ a corresponding observable transition $e_T^o$ is created in steps 8 and 14 where the source state $s$ of $e$ is replaced by $s_T$, the guard is set to $true$ and the output of the IUT transition becomes the input of the corresponding tester transition.

The processed transition $e$ of the IUT is removed from the set of outgoing transitions $E_{S_{out}}^o(s)$ (step 9). From the unprocessed set $E_{S_{out}}^o(s)$ the subset $E_{S_{out}}^o(s,a,p)$ of remaining nondeterministic transitions with the same input $a$ and a guard equivalent to $p$ is found (step 10). For each $e \in E_{S_{out}}^o(s,a,p)$ an observable transition $e_T^o$ is created (steps 12-16).

The gain functions for all controllable transitions of the tester are constructed using the structure of the tester (steps 19-21). Finally, for each controllable transition, a gain guard $p_T(V_T)$ is constructed (step 24) and the conjunction of $p_T(V_T)$ and the guard of $e_T^o$ is set to be the guard of the corresponding transition of the tester (step 25).

The details of the construction of the gain functions and gain guards is discussed in the next subsection.

An example of the tester EFMS created by Algorithm 1 is in Figure 2. The active states of the tester have the same
Algorithm 1: Build control structure of the tester

1: $E_T^e \leftarrow \emptyset; E_T^o \leftarrow \emptyset; S_T^e \leftarrow S_S; S_T^o \leftarrow \emptyset; I_T \leftarrow OS; O_T \leftarrow I_S; V_T \leftarrow V_S$
2: for all $s \in S_S$ do
3: find $E_S^{out}(s)$
4: while $E_S^{out}(s) \neq \emptyset$ do
5: get $e = (s, p, a, o, u, q)$ from $E_S^{out}(s)$
6: add $s_p$ to $S_T^p$ (passive state)
7: add $(s_p, o, a, q)$ to $E_T^p$ (controllable transition)
8: add $(s, p, o, u, q, s_p)$ to $E_T^w$ (observable trans.)
9: $E_S^{out}(s) \leftarrow E_S^{out}(s) - \{e\}$
10: if $E_S^{out}(s) = \emptyset$ then $s \leftarrow s_{passive}$
11: $E_S^{out}(s) \leftarrow E_S^{out}(s) - E_S^{out}(s_p)$
12: while $E_S^{out}(s, p) \neq \emptyset$ do
13: get $e = (s, p, a, o, u, q)$ from $E_S^{out}(s, p)$
14: add $(s_p, o, a, u, q)$ to $E_T^{o}(s, p)$ (observable trans.)
15: $E_S^{out}(s, p) \leftarrow E_S^{out}(s, p) - \{e\}$
16: end while
17: end while
18: end for
19: for all $e \in E_T^p$ do
20: construct gain function $g_e(V_T)$
21: end for
22: construct dual graph $G$ of the tester model $M_T$
23: for all $e \in E_T^p$ do
24: construct gain guard $p_{g_e}(V_T)$
25: $p \leftarrow p \land p_{g_e}(V_T)$
26: end for

Figure 2: The EFSM model of the tester for the IUT in Figure 1.

The control structure of the tester is constructed to meet the following requirements:

- The next move of the tester should be locally optimal with respect to achieving the test purpose from the current state of the tester.
- The tester should terminate after all traps are achieved or all unvisited traps are unreachable from the current state.

The gain guard evaluates to true or false at the time of the execution of the tester determining if the transition can be taken from the current state or not. The value true means that taking the transition is the best possible choice to reach some unvisited traps from the current state.

The tester makes its choice in the current state based on the structure of the tester model, the bindings of the trap variables representing the test purpose, and the current bindings of the context variables. We need some quantitative benefit measures to compare different alternative choices. For each controllable transition $e \in E_T^w$, where $E_T^w$ is the set of all controllable transitions of the tester, we define a non-negative gain function $g_e(V_T)$ that depends on the current bindings of the context variables. The gain function has the following properties:

- $g_e(V_T) = 0$, if taking the transition $e$ from the current state with the current variable bindings does not lead closer to any unvisited trap. This condition indicates that it is useless to fire the transition $e$.
- $g_e(V_T) > 0$, if taking the transition $e$ from the current state with the current variable bindings visits or leads closer to at least one unvisited trap. This condition indicates that is is useful to fire the transition $e$.
- For transitions $e_i$ and $e_j$ with the same source state, $g_{e_i}(V_T) > g_{e_j}(V_T)$, if taking the transition $e_i$ leads to an unvisited trap with smaller cost than taking the transition $e_j$. This condition indicates that it is cheaper to take the transition $e_i$ rather than $e_j$ to reach the next unvisited trap.

A gain guard for a controllable transition $e$ with the source state $s$ of the tester is defined as

$$p_{g_e}(V_T) \equiv g_e(V_T) = \max_{e_k} g_{e_k}(V_T) \land g_e(V_T) > 0, \quad (1)$$

where $g_{e_k}$ is the value of the gain function of the transition $e_k \in E_T^{out}(s)$, where $E_T^{out}(s) \subseteq E_T^p$ is the set of outgoing transitions of the state $s$. The first predicate in the logical formula (1) ensures that the gain guard is true only for a transition that leads to some unvisited trap from the current state with the highest
gain compared to the gains of the other outgoing transitions of the current state. The second predicate blocks test runs that do not serve the test purpose. The second predicate evaluates to false when all unvisited traps from the current state are unreachable or all traps are already visited. The gain guard of the tester transition enables one or more controllable transitions that should be taken at the subsequent move. If several gain functions evaluate to the same maximum value the tester selects one of the best transitions at random.

4.4 Gain Function

In this subsection we describe how the gain functions are constructed. The required properties of a gain function were specified in the previous subsection. Each transition of the IUT model is considered to have unit weight and the cost of testing is proportional to the length of the test sequence. The gain function of a transition computes a value that depends on the distance-weighted reachability of the unvisited traps from the given transition.

For the sake of efficiency we implement a heuristic in the gain function that prefers the selection of the path that visits more unvisited traps and is shorter than the alternative ones. Intuitively, in the case of two paths visiting the same number of transitions with unvisited traps and having the same lengths the path with more traps closer to the beginning of the path is preferred.

In this subsection \( M = (S, V, I, O, E) \) denotes the tester model equipped with trap variables and \( e \in E \) is a transition of the tester. We assume that the trap variable \( t \in T \) is initialised to false and set to true by the trap update function \( u_t \) associated with the transition \( e \). Therefore, reaching a trap is equivalent to reaching the corresponding transition. A transition \( e_j \) is reachable from the transition \( e_i \) if there exists a sequence of transitions \( \langle e_i, ..., e_j \rangle \), where \( e_i, e_j \in E \).

4.4.1 Shortest-Paths Tree

To find the reachable transitions from a given transition we reduce the reachability problem of the transitions to a single-source shortest-paths problem of a graph \([3]\). We construct the dual graph in a way that the vertices of the dual graph correspond to the transitions of the tester model, the shortest path of vertices in the dual graph is the shortest sequence of transitions in the tester model. Each shortest path contains only distinct vertices. Note that the shortest paths and the shortest-paths trees of a graph are not necessarily unique.

The tree \( SPT(e, G) \) represents the shortest paths from \( e \) to all reachable vertices of \( G \). We assume that the traps of the IUT model are initialised to false and a trap variable \( t \) is set to true by an update function \( u_t \) associated with the transition of the original graph. Therefore, the tree \( SPT(e, G) \) represents also the shortest paths starting with the vertex \( e \) to all reachable trap assignments. Not all transitions of the tester model contain trap variable update functions. For the evaluation of the reachability of the traps by the paths in the tree \( SPT(e, G) \) we can reduce the tree \( SPT(e, G) \) to the reduced shortest-paths tree, denoted by \( TR(e, G) \), that includes the root vertex \( e \) and only such vertices of \( SPT(e, G) \) that contain trap updates. We construct \( TR(e, G) \) by replacing those sub-paths of \( SPT(e, G) \) that do not include trap updates by hyper-edges. A hyper-edge is the shortest sub-path containing vertices without trap assignments between the root and a vertex with a trap assignment or two vertexes with trap assignments in the shortest-paths tree.

The reduced shortest-paths tree \( TR(e, G) \) contains the shortest paths beginning with the transition \( e \) to all reachable traps in the dual graph \( G \) (and thus also in the tester model). In the reduction of the shortest-paths tree \( SPT(e, G) \) to \( TR(e, G) \) we label each vertex that contains a trap variable update \( u_t \) by the corresponding trap \( t \) and
replace each sub-path containing vertices without trap updates by a hyper-edge \((t_i, w, t_j)\) where \(t_i\) is the label of the source vertex, \(t_j\) is the label of the destination vertex and \(w\) is the length of the sub-path. Also, during the reduction we remove those sub-paths (hyper-edges) that end in the leaf vertices of the tree that do not contain any trap variable updates.

Figure 4 (left) shows the shortest-paths tree \(SPT(e_{01}, G)\) with the root vertex \(e_{01}\) for the dual graph in Figure 3. For example, the path \((e_{01}, e_{14}, e_{26}, e_{06})\) from the root vertex \(e_{01}\) to the vertex \(e_{06}\) in the shortest-paths tree in Figure 4 is the shortest sequence of transitions beginning with the transition \(e_{01}\) that reaches \(e_{06}\) in the example of the tester model in Figure 2. The reduced shortest-paths tree

![Image](image_url)

**Figure 4:** The shortest-paths tree \(SPT(e_{01}, G)\) (left) and the reduced shortest-paths tree \(TR(e_{01}, G)\) (right) from the transition \(e_{01}\) of the graph shown in Figure 3.

\(TR(e_{01}, G)\) from the vertex \(e_{01}\) to the reachable traps for the dual graph in Figure 3 is represented in Figure 4 (right). All vertices except the root of the reduced shortest-paths tree \(TR(e_{01}, G)\) are labelled with the trap variables, and the hyper-edges between the vertices are labelled with the number of transitions in the sub-path the hyper-edge represents. The tree \(TR(e_{01}, G)\) contains the shortest paths beginning with the transition \(e_{01}\) to all traps in the tester model in the Figure 2. For example, the tree \(TR(e_{01}, G)\) shows that there exists a path beginning with the transition \(e_{01}\) to the trap \(t_6\), and this path visits traps \(t_1\) and \(t_4\) on the way.

### 4.4.2 Algorithm of Constructing the Gain Function

The return type of the gain function is non-negative rational \(\mathbb{Q}^+\). That follows explicitly from the construction rules of the gain function (see steps below) and from that the corpus of rational numbers is closed under addition and the \(\max\) operator. The algorithm of construction of the gain function for the transition \(e\) of the tester automaton \(M\) from the dual graph \(G\) is the following:

1. Construct the shortest-paths tree \(SPT(e, G)\) for the transition \(e\) of the dual graph \(G\).

2. Reduce the shortest-paths tree \(SPT(e, G)\) as described in subsection 4.4.1. The reduced tree is denoted by \(TR(e, G)\). Assign the length of the sub-path of each hyper-edge \((t_i, w, t_j)\) of \(TR(e, G)\) to \(w\).

3. Represent the reduced tree \(TR(e, G)\) as a set of elementary sub-trees of height 1 where each elementary sub-tree is specified by the production rule of the form
   \[
   \nu_i \rightarrow \big| j \in \{1, \ldots, k\} \nu_j, 
   \]  
   where the non-terminal symbol \(\nu_i\) denotes the root vertex of the sub-tree and each \(\nu_j\) (where \(j \in \{1, \ldots, k\}\)) denotes a leaf vertex of that sub-tree, and where \(k\) is the branching factor. \(\nu_0\) corresponds to the root vertex \(e\) of the reduced tree.

4. Rewrite the right-hand sides of the productions constructed in step 3 as arithmetic terms, thus getting the production rule in the form
   \[
   \nu_i \rightarrow (-t_i)^7 \frac{c}{d(v_0, \nu_i) + 1} + \max_{j=1,k} (\nu_j),
   \]  
   where \(t_i\) denotes the trap variable \(t_i\) lifted to type \(\mathbb{N}\), \(c\) is a constant for the scaling of the numerical value of the gain function, and \(d(v_0, \nu_i)\) the distance between vertices \(v_0\) and \(\nu_i\) in the labelled tree \(TR(e, G)\). The distance is defined by the formula
   \[
   d(v_0, \nu_i) = 1 + \sum_{j=1} l w_j
   \]  
   where \(l\) is the number of hyper-edges on the path between \(v_0\) and \(\nu_i\) in \(TR(e, G)\) and \(w_j\) is the value of the label \(w\) corresponding to the concrete hyper-edge.

5. For each symbol \(\nu_i\) denoting a leaf vertex in \(TR(e, G)\) define a production rule:
   \[
   \nu_i \rightarrow (-t_i)^7 \frac{c}{d(v_0, \nu_i) + 1}
   \]  
   6. Apply the production rules (3) and (4) starting from the root symbol \(\nu_0\) of \(TR(e, G)\) until all non-terminal symbols \(\nu_i\) are substituted with the terms that include only terminal symbols \(t_i\) and \(d(v_0, \nu_i)\), \((i \in \{0, \ldots, n\})\), where \(n\) is the number of trap variables in \(TR(e, G)\). The root vertex \(\nu_0 = e\) of the labelled tree \(TR(e, G)\) may not have a trap label. Instead of a trap variable \(t_i\), use a constant \(true\) as the label resulting \((-true)^7 = 0\) in the rule (3).

It has to be pointed out that the gain function characterises the expected gain only within the planning horizon. The planning horizon is determined by the lengths of the paths in reduced shortest-paths tree.

Table 1 shows the results of the application of the production rules (2), (3) and (4) to the vertices of the reduced shortest-paths tree \(TR(e_{01}, G)\) in Figure 4 (right). As the root \(e_{01}\) is not labelled with a trap variable, the transition \(e_{01}\) does not update any trap, a constant \(true\) is used in the production rule (3) in the place of the trap variable resulting \((-true)^7 = 0\) in the first row of Table 1. Application of the production rules (3) and (4) to the tree \(TR(e_{01}, G)\) starting from the root vertex \(e_{01}\) results in the gain function given in the first row of Table 2. Table 2 presents the gain functions for the controllable transitions of the tester model (Figure 2). The gain guards for all controllable transitions of the tester model are given in Table 3. The type lifting functions of the traps have been omitted from the tables for the sake of brevity.
4.5 Complexity of Constructing the Tester

The complexity of the synthesis of the reactive planning tester is determined by the complexity of the construction of the gain functions. For each gain function the cost of finding the shortest-paths tree for a given transition in the dual graph by breadth-first-search is \(O(|\mathcal{V}_D| + |\mathcal{E}_D|)\) [3], where \(|\mathcal{V}_D| = |\mathcal{E}_D|\) is the number of transitions and \(|\mathcal{E}_D|\) is the number of transition pairs of the tester model. The number of transition pairs of the tester model is mainly defined by the number of transition pairs of the observable and controllable transitions which is bounded by \(|\mathcal{E}_S|^2\). For all controllable transitions of the tester the upper bound of the complexity of the offline computations of the gain functions is \(O(|\mathcal{E}_S|^3)\).

At runtime each choice by the tester takes no more than \(O(|\mathcal{E}_S|^2)\) arithmetic operations to evaluate the gain functions for the outgoing transitions of the current state.

5. EXECUTING THE TESTER MODEL

The tester model can be transformed to any programming language and executed against the IUT. It includes enough information that allows the tester to run the test with the nondeterministic IUT. The tester model knows how to stimulate the IUT in each active state of the tester in order to complete the test run successfully. Meeting a test purpose is equivalent to having visited all traps. A test run is started with the goal to visit all specified traps. Execution of the tester starts from the initial state of the tester model with the initial values of the context variables. If the current state of the tester is passive then the tester observes the output of the IUT and takes a transition that matches the output of the IUT. If the current state is active then the tester evaluates the gain guards of the outgoing transitions of the current state and takes a transition where the gain guard evaluates to \(true\). If there is more than one such transition then one of them is selected randomly. This can happen
only if the gain functions of the transitions return an equal gain value meaning that such alternative choices are equally good. If in the current state of the tester no unvisited traps are reachable then all the gain functions evaluate to zero, the gain guards become disabled and the tester run terminates.

In the case of a nondeterministic IUT the coverage of all traps depends on the nondeterministic choices of the IUT. Under the fairness assumption (all nondeterministic transitions are eventually chosen) and when all traps are reachable, the test run will eventually terminate. In practice the testing time is limited and therefore a test duration limit is specified in addition to the termination condition where all traps should be visited. As it is allowed that the IUT model be not strongly connected then the IUT may get to a state during a test run where reaching the rest of the unvisited traps is impossible. In such case the IUT must be reset and the execution of the tester model is restarted to visit the remaining unvisited traps. The procedure is repeated until all traps are visited or the limiting test time has passed.

6. EXPERIMENTAL RESULTS

Table 4 and Table 5 present results of experiments with three models. The column labelled Model 1 corresponds to the example shown in Figure 1. The Model 2 is constructed by composing two copies of the example model in Figure 1 where state s2 of the first copy is merged with the state s1 of the second one. The Model 3 is a composition of four copies of the Model 1 merging states s2 and s1 of the consecutive copies. The Model 2 (Model 3) has 5 (9) states and 16 (32) transitions with 4 (8) pairs of nondeterministic transitions. We have implemented the tester models of the examples using UPPAAL-TIGA [1]. Nondeterministic choices were simulated in UPPAAL-TIGA which randomly selects one of the multiple enabled transitions.

Comparison of the efficiency in terms of test sequence lengths of the testers running by random choice, anti-ant and reactive planning algorithms are given in Table 4 and Table 5. The tables give results in the form average ± standard deviation of 30 experiments. The minimal possible lengths of the test sequences to reach all transitions in Model 1, Model 2 and Model 3 are 12, 24, and 48 inputs, respectively. Table 5 shows the average lengths of the test sequences satisfying the test purpose to cover a single transition which was selected to be the farthest from the initial state. The minimal possible lengths of the test sequences to reach the single selected transition in the models are 3, 4, and 6 inputs, respectively.

The experiment shows that for a test purpose to cover all transitions the reactive planning tester results on average in many times shorter test sequence than the random choice tester with considerably lower standard deviation. The advantage of the reactive planning tester compared to the anti-ant algorithm becomes more articulated when there are more nested loops present in the models (as in Model 2 and Model 3 compared to Model 1).

If the test purpose is to cover only selected transitions, the reactive planning tester outperforms the tester that uses the anti-ant algorithm. This is caused by the fact that due to reactive planning the choices of the tester are explicitly directed towards fulfilling the test purpose.

We have also experimented with deterministic IUT models. These results show that when the IUT model is deterministic, the reactive planning tester achieves the test purpose with a test sequence that has length close to minimal.

Our selection of experiments is in no way exhaustive but the results match our expectations in the concrete cases. Manual construction of larger examples is infeasible without tool support.

7. CONCLUSION

In this paper we proposed a model-based construction of an online tester for black-box testing of the IUT. The IUT is modelled in terms of an output observable nondeterministic EFSM with the assumption that all transition paths are feasible. A test purpose is attributed to the IUT model by a set of Boolean variables called traps.

The main contribution of our work is an algorithm how to construct a tester that at runtime selects a suboptimal test path from trap to trap by finding the shortest path to the next unvisited trap in each iteration. The principles of reactive planning are implemented in the form of the rules of selecting the shortest paths at runtime. The rules are constructed in advance from the IUT model and the test purpose. The rules are encoded into the guards of the transitions of the tester and are evaluated by the tester in every state when the selection between alternative outgoing transitions should be made. Any costly model exploration and path finding operations are not needed online.

In the context of our examples, the reactive planning tester is more efficient at run time than random choice and anti-ant algorithms. The planning feature of the reactive planner results in significantly shorter average test sequence lengths. The reactive planner outperforms the anti-ant algorithms in cases where more directed search is presumed, i.e. the test purpose covers the model partially.

The next step in evaluating the applicability of the reactive planning tester on further case studies requires an implementation of tool support that is part of the future work. In this work we made the assumption that all transition paths in the EFSM are feasible but future work involves removing the feasible paths assumption.
Acknowledgements

We thank the anonymous referees for their insightful comments. This work was partially supported by the Estonian Science Foundation under grant No 5775, by ELIKO Competence Centre project “Integration Platform for Development Tools of Embedded Systems”, by the Estonian Doctoral School in Information and Communication Technology, and Tiigriülikool+ programme of the Estonian Information Technology Foundation.

8. REFERENCES


