

Requirement-Based Testing: Enhancing Reinforcement Learning with Game Theory

Ocan Sankur^{*}, Thierry Jéron^{*}, Nicolas Markey^{*}, David Mentré[†], Reiya Noguchi[‡]

^{*}Univ Rennes, Inria, CNRS, Irista, Rennes, France

[†]Mitsubishi Electric R&D Centre Europe, Rennes, France

[‡]Mitsubishi Electric Corporation, Tokyo, Japan



Abstract—We consider the automatic online synthesis of black-box test cases from functional requirements specified as automata for reactive implementations. The goal of the tester is to reach some given state, to satisfy a coverage criterion, while monitoring the violation of the requirements. We develop an approach based on Monte Carlo Tree Search (MCTS), which is a classical technique in reinforcement learning for computing optimal policies. Seeing the automata requirements as a game between the implementation and the tester, we develop a heuristic by biasing the search towards inputs that are promising in this game. We experimentally show that our heuristic accelerates the convergence of MCTS, thus improving the performance of testing.

I. INTRODUCTION

Requirement engineering and testing are two important and related phases in the development process. Indeed, test cases are usually derived from functional specifications and documented with the requirements they are supposed to check. Several existing tools allow one to automatically generate test cases from formal functional specifications and/or formal requirements (see e.g., the tools T-VEC [1], TorXakis [2] or Stimulus [3], and surveys [4] and [5]).

We are interested in black-box testing reactive implementations w.r.t. automata requirements. Reactive implementations are programs that alternate between reading an input valuation and writing an output valuation, which are typical, e.g. of controllers for manufacturing systems [6]. Requirements are given as automata recognizing valid sequences of valuations of input and output variables. The *conformance* of an implementation to a set of requirements is formalized as the absence of input-output valuation sequences generated by the implementation that are rejected by the automaton. Black-box testing means that the tester does not have access to the source code, and only interacts with the system via feeding inputs and observing outputs [7].

The goal of testing is to drive the implementation to some particular state or show some particular behavior where non-conformance is suspected. These are described by *test objectives*. They are typically derived from requirements or from coverage criteria, e.g. state or transition coverage. We give several examples throughout the paper.

There are several black-box testing algorithms and tools in the literature. The closest to our approach are the tools TorX

and TorXakis [8], [2], [9] which are based on the *iooco* testing theory [10]. These allow the user to specify the automata-based requirements reading input-output valuations as well as test objectives (called test purposes) in a process algebra language. They perform tests by picking random input valuations, and observing the outputs from the implementation, while checking for non-conformance. Because tests are performed using random walks in this approach, deep traces satisfying the test objective or violating the requirements are hard to find in practice.

This issue has been addressed in many works by interpreting the test synthesis in a *reinforcement learning*(RL) setting [11]. Reinforcement learning is a set of techniques for computing strategies that optimize a given reward function based on interactions of an agent with its environment. Here testing is seen as a game between the tester and the implementation: the former player selects inputs, and the latter player selects outputs. Because the implementation is black-box, the tester is playing an unknown game, but can discover the game through interactions. Using a game approach for test synthesis has long been advocated [12]; and online testing for interface automata were considered in [13] using RL.

Because reaching a test objective is a 0/1 problem (an execution either reaches the objective and has a reward 1, or does not reach it and has reward 0), RL algorithms are usually very slow in finding deep traces. The application of RL to black-box testing thus requires the use of *reward shaping* [14] which consists in assigning intermediate rewards to steps before the objective is reached; these are used to guide the search to more promising input sequences and can empirically accelerate convergence. Reward shaping has been used for testing, e.g., in [15] where an RL algorithm (Q-learning) was used for testing GUI applications with respect to linear temporal logic (LTL) specifications; rewards were then computed based on transformations on the target LTL formula. A similar approach was applied for testing distributed algorithm implementations in [16] by manually specifying *waypoints* which are steps the test algorithm must reach to reach the objective. We focus here on improving RL-based test algorithms w.r.t. automata-based requirements, which, combined with reward shaping and the heuristic we develop here, as a systematic way of obtaining such waypoints *automatically* to guide the search.

Contributions Our target is to improve the performance of black-box online testing based on reinforcement learning. More precisely, we develop heuristics for a Monte Carlo Tree Search (MCTS) algorithm applied in this setting, based on a game-theoretic analysis of the requirement automaton, combined with an appropriate reward shaping scheme.

Monte-Carlo Tree Search [17] (see also [18]) is a RL technique to compute optimal policies in games. It consists in exploring the available moves randomly, while estimating the average reward of each newly-explored move and updating the reward estimates of previously selected moves. More precisely, it uses a so-called *tree policy* to select which branch of the tree to explore, and a *roll-out policy* to evaluate the reward of a branch.

Our main contribution is a heuristic for biasing both the tree policy and the roll-out policy in MCTS in order to reach test objectives faster, while maintaining convergence guarantees in the limit. The heuristic is based on a *greedy* test strategy we introduce, defined as follows. We first see the requirement automaton as a turn-based game between the tester (selecting inputs) and the implementation (selecting outputs). We compute the winning region using standard algorithms [19]: inside this region, the tester can ensure reaching the test objective. Outside of this region, we compute *cooperative* strategies which guarantee reaching the test objective provided that the implementation selects a particular output in one step. We alternate the computation of winning regions and cooperative states to cover the full reachable (and coreachable) state space. Our game-based greedy test strategy consists in selecting uniformly at random inputs that are part of a winning strategy if any, or allow the implementation to cooperate.

Our novel testing algorithm consists in integrating the greedy strategy inside the tree and roll-out policies. The probability of using the greedy strategy vanishes as time progresses, so the algorithm is equivalent to the standard MCTS in the limit; inheriting its convergence properties.

We present experimental results on two distributed algorithms and an implementation of a planning problem to show the effectiveness of our approach in comparison with a basic MCTS solution and uniform random testing.

II. PRELIMINARIES

We fix a set of atomic propositions $\mathbf{AP}$, partitioned into $\mathbf{AP}^{\text{in}} \uplus \mathbf{AP}^{\text{out}}$, that represent Boolean input- and output variables of the system. A *valuation* of $\mathbf{AP}$ is an element ν of $2^{\mathbf{AP}}$ determining the set of atomic propositions which are true (or equivalently, it is a mapping $\nu: \mathbf{AP} \rightarrow \{\top, \perp\}$). We denote by ν^{in} (respectively ν^{out}) the projection of ν on $\mathbf{AP}^{\text{in}}$ (resp. $\mathbf{AP}^{\text{out}}$) such that $\nu = \nu^{\text{in}} \uplus \nu^{\text{out}}$. We write $\mathcal{B}(\mathbf{AP})$ for the set of Boolean combinations of atomic propositions in $\mathbf{AP}$. That a valuation ν satisfies a formula $\phi \in \mathcal{B}(\mathbf{AP})$, denoted by $\nu \models \phi$, is defined in the usual way.

In considered reactive systems, at each step, the environment first sets an input valuation ν^{in} , then the system immediately sets an output valuation ν^{out} . The valuation observed at this step is thus $\nu = \nu^{\text{in}} \uplus \nu^{\text{out}}$. A *trace* of the system is a sequence $\sigma = \nu_1 \cdot \nu_2 \cdots \nu_n$ of input and output valuations.

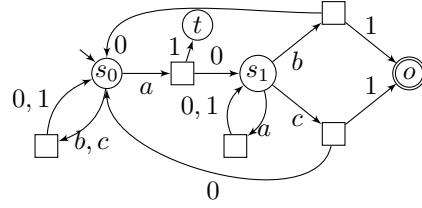


Fig. 1. Example of a (deterministic) automaton expressing requirements.

a) *Automata*: We use automata to express requirements, and as models for the implementations under test. When considered as requirements, they monitor the system through the observation of the values of the Boolean variables. Transitions of automata are guarded with Boolean constraints on $\mathbf{AP}$ that need to be satisfied for the automaton to take that transition. For convenience, we handle input- and output valuations separately. Formally,

Definition 2.1: An *automaton* is a tuple $\mathcal{A} = \langle S = S^{\text{in}} \uplus S^{\text{out}}, s_{\text{init}}, \mathbf{AP}, T, F \rangle$ where S is a finite set of states, $s_{\text{init}} \in S^{\text{in}}$ is the initial state, $T \subseteq (S^{\text{in}} \times \mathcal{B}(\mathbf{AP}^{\text{in}}) \times S^{\text{out}}) \uplus (S^{\text{out}} \times \mathcal{B}(\mathbf{AP}^{\text{out}}) \times S^{\text{in}})$ is a finite set of transitions, and $F \subseteq S$ is the set of accepting states.

For two states s^{in} and s'^{in} and a valuation ν , we write $s^{\text{in}} \xrightarrow{\nu} s'^{\text{in}}$ when there exist a state s^{out} and transitions $(s^{\text{in}}, g^{\text{in}}, s^{\text{out}})$ and $(s^{\text{out}}, g^{\text{out}}, s'^{\text{in}})$ in T such that $\nu^{\text{in}} \models g^{\text{in}}$ and $\nu^{\text{out}} \models g^{\text{out}}$.

For a trace $\sigma = \nu_1 \cdot \nu_2 \cdots \nu_n$ in $(2^{\mathbf{AP}})^*$, we write $s^{\text{in}} \xrightarrow{\sigma} s'^{\text{in}}$ if there are states $s_0^{\text{in}}, s_1^{\text{in}}, \dots, s_n^{\text{in}}$ such that $s_0^{\text{in}} = s^{\text{in}}$, $s_n^{\text{in}} = s'^{\text{in}}$, and for all $i \in [1, n]$, $s_{i-1}^{\text{in}} \xrightarrow{\nu_i} s_i^{\text{in}}$. A trace $\sigma \in (2^{\mathbf{AP}})^*$ is accepted by $\mathcal{A}$ if $s_{\text{init}} \xrightarrow{\sigma} s$ for some $s \in F$. We denote by $\text{Tr}(\mathcal{A})$ the set of accepted traces.

An automaton is *input-complete* if from any (reachable) state s^{in} and any valuation $\nu^{\text{in}} \in 2^{\mathbf{AP}^{\text{in}}}$, there is a transition $(s^{\text{in}}, g^{\text{in}}, s^{\text{out}})$ in T such that $\nu^{\text{in}} \models g^{\text{in}}$. It is *output-complete* if a similar requirement holds for states in S^{out} and valuations in ν^{out} , and it is *complete* if it is both input- and output-complete. An automaton is *deterministic* when, for any two transitions (s, g_1, s_1) and (s, g_2, s_2) issued from a same source s , if $g_1 \wedge g_2$ is satisfiable, then $s_1 = s_2$.

We are mainly interested in S^{in} , since states in S^{out} are intermediary states that help us distinguish input and output valuations. For a state s^{in} and a valuation ν , we let $\text{Post}_{\mathcal{A}}(s^{\text{in}}, \nu)$ denote the set of states s'^{in} such that $s^{\text{in}} \xrightarrow{\nu} s'^{\text{in}}$, and $\text{Pre}_{\mathcal{A}}(s^{\text{in}}, \nu)$ denote the set of states s'^{in} such that $s'^{\text{in}} \xrightarrow{\nu} s^{\text{in}}$. Notice that for deterministic complete automata, $\text{Post}_{\mathcal{A}}(s^{\text{in}}, \nu)$ is a singleton.

The set of *immediate predecessors* $\text{Pre}_{\mathcal{A}}(B)$ of a set $B \subseteq S^{\text{in}}$, and the set of its *immediate successors* $\text{Post}_{\mathcal{A}}(B)$ are defined respectively as $\text{Pre}_{\mathcal{A}}(B) = \bigcup_{s^{\text{in}} \in B, \nu \in 2^{\mathbf{AP}}} \text{Pre}_{\mathcal{A}}(s^{\text{in}}, \nu)$, and $\text{Post}_{\mathcal{A}}(B) = \bigcup_{s^{\text{in}} \in B, \nu \in 2^{\mathbf{AP}}} \text{Post}_{\mathcal{A}}(s^{\text{in}}, \nu)$.

From these sets, one can define the set of states from which B is reachable (i.e., that are co-reachable from B), as $\text{Pre}_{\mathcal{A}}^*(B) = \text{lfp}(\lambda X. (B \cup \text{Pre}_{\mathcal{A}}(X)))$, and the set of states that are reachable from B , $\text{Post}_{\mathcal{A}}^*(B) = \text{lfp}(\lambda X. (B \cup \text{Post}_{\mathcal{A}}(X)))$, where lfp denotes the least-fixpoint operator.

Safety automata form a subclass of automata having a distinguished set $\mathbf{Err} \subseteq S^{\text{in}}$ of error states that are non-accepting and absorbing (i.e., no transitions leave $\mathbf{Err}$), and complement the set F of accepting states in S (i.e., $F = S \setminus \mathbf{Err}$).

The product of two automata is defined by combining pairs of transitions and defining the guard of the new transition as the conjunction of the guards of the two transitions (See Appendix A).

For the sake of readability, in Figure 1, we use input- and output letters instead of atomic propositions. Here, $\{a, b\}$ are input letters, and $\{0, 1\}$ are output letters. This automaton is deterministic; moreover, setting $\mathbf{Err} = \{t\}$ makes it a safety automaton. It could be made complete by adding looping input-output transitions (similar to the transitions to the bottom left of s_0) on t and o .

III. TESTING FROM REQUIREMENTS

We want to use testing to check whether a system implementation satisfies its requirements. We thus formalize a notion of conformance to a set of requirements.

We identify a requirement with its complete deterministic safety automaton, and write R for both. For a set $\mathcal{R} = \{R_i\}_{i \in I}$ of requirements, each specified by an automaton R_i , we denote by $\mathcal{A}_{\mathcal{R}}$ the product automaton $\otimes_{i \in I} R_i$.

For any requirement R defined by a complete deterministic automaton, and any finite trace σ , we write σ **fails** R if running σ in R from its initial state enters its error set $\mathbf{Err}_R$. For a trace σ and a set of requirements $\mathcal{R}$ we write σ **fails** $\mathcal{R}$ to mean that σ **fails** $\mathcal{A}_{\mathcal{R}}$.

We want to test a system against a set of requirements $\mathcal{R}$. We consider a deterministic system implementation I (the implementation under test), producing Boolean traces in $(2^{\text{AP}})^*$. We assume that this system is a black box that proceeds as follows: at each step, an input valuation in $2^{\text{AP}^{\text{in}}}$ is provided to the system by the tester, and the system answers by producing an output valuation in $2^{\text{AP}^{\text{out}}}$ (on which the tester has no control). We consider deterministic implementations with the following assumptions. I behaves as an unknown finite automaton over AP ; it is *input-complete*, meaning that any valuation in $2^{\text{AP}^{\text{in}}}$ can be set by the tester, and from any input state of I , it is possible to reset I to the unique initial state at any time. Furthermore, I is *output-deterministic*, i.e. any state in S^{out} has exactly one transition. This ensures that if one feeds the implementation with an input sequence from the initial state, then the system produces a unique output sequence. We denote by $\mathcal{I}$ the class of all such implementations producing traces in $(2^{\text{AP}})^*$. The behaviour of I is characterized by the set of all traces produced by the interaction between the tester and the system. Denote by $\text{Tr}(I)$ the set of traces produced by I .

We say that a system implementation $I \in \mathcal{I}$ *conforms* to a set of requirements $\mathcal{R}$ on AP if for all $\sigma \in \text{Tr}(I)$, it is not the case that σ **fails** $\mathcal{R}$.

Appendix B contains examples of requirements and corresponding automata.

a) *Test Objectives.*: In testing practice, each test case is related to a particular goal, e.g., derived from a coverage

criterion. In our case, test objectives are given by requirement automata (as in [20]), and the goal is to cover the accepting states.

Formally, consider a set $\mathcal{R}$ of requirements, and let $S_{\mathcal{R}}$ denote the state space of $\mathcal{A}_{\mathcal{R}}$. A *test objective* is a set of states $O \subseteq S_{\mathcal{A}_{\mathcal{R}}}^{\text{in}}$. A trace σ *covers* a test objective O , if the unique execution of $\mathcal{A}_{\mathcal{R}}$ on σ ends in O .

The problem that we address in the rest of the paper is the following:

Input: requirement set $\mathcal{R}$, test objective O , and deterministic implementation I ;

Output: a trace of I that covers O , if such a trace exists.

An algorithm that solves this problem is called a *test algorithm*. A test algorithm is *complete* if for any input $\mathcal{R}, O$, and I that contains a trace that covers O , it returns a trace covering O . It is *almost-surely complete* if in such a case, it returns a trace covering O with probability 1.

The covering traces we are looking for are thus traces of I that satisfy a given objective. These witness the fact that we have met a particular coverage criterion. While our testers focus on satisfying a given objective, they always monitor for errors: if the trace fails $\mathcal{R}$ then we stop and report this as an error. One can of course define $\mathbf{Err}$ as a test objective; in this case, the testing process will focus on finding errors.

A. Baseline Test Algorithms

Consider a set $\mathcal{R}$ of requirements, specified by a deterministic complete automaton $\mathcal{A}_{\mathcal{R}} = \langle S_{\mathcal{R}}, s_{\text{init}}^{\mathcal{R}}, \text{AP}, T_{\mathcal{R}}, F_{\mathcal{R}} \rangle$, and an implementation modeled by an input-complete, output-deterministic finite automaton $I = \langle S_I, s_{\text{init}}^I, \text{AP}, T_I, F_I \rangle$. Recall that in our setting, the set $\mathcal{R}$ of requirements, thus also its automaton $\mathcal{A}_{\mathcal{R}}$, is known, while the implementation I is considered to be black box to the tester.

Consider a particular test objective $O \subseteq S_{\mathcal{A}_{\mathcal{R}}}^{\text{in}}$. Let $\text{CoReach}(\mathcal{A}_{\mathcal{R}}, O) = \text{Pre}_{\mathcal{A}_{\mathcal{R}}}^*(O)$ denote the set of input states of $\mathcal{A}_{\mathcal{R}}$ from which O is reachable.

Our aim is to design online testing algorithms that compute inputs to be given to the implementation I in order to generate a trace that either covers O , or detects non-conformance by reaching an $\mathbf{Err}$ state (or both if O contains $\mathbf{Err}$ states); notice that since I is output-deterministic, each such input sequence defines a unique trace of I . The testing process runs as follows: from a state $s^{\mathcal{A}_{\mathcal{R}}}$ of $\mathcal{A}_{\mathcal{R}}$ and a state s^I of I , the test algorithm returns an input valuation ν^{in} ; this input valuation is fed to the implementation, which returns an output valuation ν^{out} and moves to a new state t^I ; the resulting valuation $\nu^{\text{in}} \cup \nu^{\text{out}}$ moves the automaton $\mathcal{A}_{\mathcal{R}}$ from $s^{\mathcal{A}_{\mathcal{R}}}$ to a new state $t^{\mathcal{A}_{\mathcal{R}}}$. The process then continues from $t^{\mathcal{A}_{\mathcal{R}}}$ and t^I , until reaching a test objective or $\mathbf{Err}$.

We write $\mathcal{A}_{\mathcal{R}} \otimes I$ for the synchronized product of $\mathcal{A}_{\mathcal{R}}$ and I : this is a deterministic automaton, of which we observe only the first component (i.e. the state of $\mathcal{A}_{\mathcal{R}}$), while we have no information and no observation concerning the second component except from the produced outputs. Our aim is to build a tester to cover some objectives in this *partially-observable* deterministic automaton. An example of the product $\mathcal{A}_{\mathcal{R}} \otimes I$ built for the automaton of Fig. 1 is given in Appendix B.

Since we do not know I , each input valuation ν^{in} should be selected only based on the trace generated so far, and possibly based on information collected on previous attempts.

Before explaining our test algorithms, we introduce some vocabulary to describe the configuration where the testing process ends. Assume that we have generated a trace σ by interacting with I from its initial state. Let s_σ denote the state of $\mathcal{A}_\mathcal{R}$ reached after reading σ from the initial state. Four cases may occur: if $s_\sigma \in O$, then σ is a *covering trace* for O ; if $s_\sigma \in \mathbf{Err}_{\mathcal{A}_\mathcal{R}}$, then σ is an *error trace*; if $s_\sigma \notin \text{CoReach}(\mathcal{A}_\mathcal{R}, O)$, then σ is *inconclusive*; otherwise, $s_\sigma \in \text{CoReach}(\mathcal{A}_\mathcal{R}, O) \setminus O$ and σ is *active*.

Intuitively, in the first case, we have found the desired covering trace, and we can stop. In the second case, we have found a trace failing one of the requirements of $\mathcal{A}_\mathcal{R}$, and we can also stop: the implementation does not conform to $\mathcal{A}_\mathcal{R}$. Notice that these two cases are not exclusive since we can have $O \cap \mathbf{Err}_{\mathcal{A}_\mathcal{R}} \neq \emptyset$. In the third case, no matter how we extend σ , we will never cover O ; so the tester should stop and start again by resetting I . In the last case, σ is active in the sense that it might still be possible to try to extend σ to reach the objective.

It should be clear that σ being active (last case) does not mean that O is reachable from the corresponding state (s_σ, s^I) of the product $\mathcal{A}_\mathcal{R} \otimes I$, as this depends on the (unknown) implementation I being considered: states in $\text{CoReach}(\mathcal{A}_\mathcal{R}, O)$ are those for which *some* implementation in $\mathcal{I}$ can reach O . We illustrate this scenario in Example A.3 in the appendix.

1) *Naive Uniform Testing*: The simplest test algorithm implemented in tools such as TorXakis [9] consists in picking inputs at random at each step. We define here an improved variant. Let $\text{CoReach}^{\text{in}}(\mathcal{A}_\mathcal{R}, O)$ denote the set of pairs (s, ν^{in}) where s is a state of $\mathcal{A}_\mathcal{R}$, and ν^{in} is an input valuation for which there exists some output valuation ν^{out} such that $\text{Post}_{\mathcal{A}_\mathcal{R}}(s, \nu) \in \text{CoReach}(\mathcal{A}_\mathcal{R}, O)$, where $\nu = \nu^{\text{in}} \cup \nu^{\text{out}}$. In fact, after observing trace σ ending in state s of $\mathcal{A}_\mathcal{R}$, the tester has no reason to give an input ν^{in} such that $(s, \nu^{\text{in}}) \notin \text{CoReach}^{\text{in}}(\mathcal{A}_\mathcal{R}, O)$: such an input would lead to a trace that is inconclusive, with no hope of reaching O .

A very simple test algorithm is the following: starting from the initial state of the implementation, we store in s the initial state of $\mathcal{A}_\mathcal{R}$. As long as the trace being produced is active, we select uniformly at random an input valuation among $\{\nu^{\text{in}} \mid (s, \nu^{\text{in}}) \in \text{CoReach}^{\text{in}}(\mathcal{A}_\mathcal{R}, O)\}$. We observe the output ν^{out} given by I , and update s as $\text{Post}_{\mathcal{A}_\mathcal{R}}(s, \nu^{\text{in}} \cup \nu^{\text{out}})$. There are three cases when this process stops: if $s \in O$, then we stop and report the generated trace as a covering trace for O ; if $s \in \mathbf{Err}_{\mathcal{A}_\mathcal{R}}$, we also stop and report a failure; if the current trace is inconclusive, then we reset I , set s to the initial state of $\mathcal{A}_\mathcal{R}$, and start again.

Let this test algorithm be called *uniform*. For any bound K , let uniform_K be the uniform testing algorithm in which we stop each run after K steps, so that the generated traces have length at most K .

Lemma 3.1: For each requirement set $\mathcal{R}$, implementation I , and test objective O , there exists $K > 0$ such that if O is

reachable in $\mathcal{A}_\mathcal{R} \otimes I$, then uniform_K finds a covering trace with probability 1.

Note that there is no need to fix K . If K is increased towards infinity, then a covering trace is found with probability 1. Furthermore, the uniform distribution can also be relaxed: any algorithm that picks each input of $\{\nu^{\text{in}} \mid (s, \nu^{\text{in}}) \in \text{CoReach}^{\text{in}}(\mathcal{A}_\mathcal{R}, O)\}$ with probability at least some fixed $\epsilon > 0$ has this property.

2) *Testing Based on Reinforcement Learning*: The online-testing problem can be seen as a reinforcement-learning (RL) problem as follows. The considered system is the implementation I , seen as a one-player deterministic game. The goal is to find a sequence of inputs that guides the system to a given objective. We assign a reward to each trace: a covering trace has reward 0, other traces have reward 1.

Reinforcement learning is a set of techniques that can be used to learn strategies that maximize reward / minimize cost in games [11]. In this paper, we use Monte Carlo Tree Search [17], [21]. This is an algorithm explores available moves randomly, while estimating the potential of each newly-explored move and updating the potential of previously selected moves.

More precisely, MCTS builds a weighted tree of possible plays of the game iteratively as follows:

- **Selection:** from the root of the tree, select moves, using a *tree policy*, until reaching a node where some move has not been explored;
- **Expansion:** add a new child corresponding to that move;
- **Simulation:** simulate a random play, using a *roll-out policy*, from that new child;
- **Propagation:** assign the reward of that play to the new child, and update the rewards of its ancestors accordingly.

Different tree policies can be used to pick the successive moves during the selection phase, based on statistics obtained from previous iterations. We use UCT (Upper Confidence bounds applied to Trees) [21] which is standard in many applications.

Once an unvisited action has been selected and the tree has been expanded with a new node, a roll-out policy is applied to evaluate the potential of that new action, usually by randomly selecting inputs that form a path from the resulting configuration. This evaluation gives a first reward to the newly created node, which is back-propagated to all its predecessors in the tree; each node of the tree stores statistics from previous rounds, including the number of visits and its average reward.

In the limit, the procedure is guaranteed to provide the optimal expected reward values from each state and move. In practice, the procedure can be interrupted at any time (depending on the available resources devoted to the search), and the current best moves from all states of the tree provide a strategy.

In our case, each simulation is bounded by K steps. Such a bound is necessary since some simulations might never reach the objective, an error, or an inconclusive state and thus never terminate. We use uniform_K (Section III-A1) as the roll-out policy. Choosing the inputs uniformly in the simulation phase

is standard. Here, we simply improve this by restricting the inputs to remain in the coreachable set.

a) *Reward Shaping: Accelerating Convergence:* One technique that is used to help reinforcement-learning algorithms converge faster is *reward shaping* [14], which consists in assigning real-valued rewards to traces, to give more information than just the binary 0/1. For instance, if the trace induced a run in $\mathcal{A}_{\mathcal{R}}$ that became very close to the objective, then it might be given a better (lower) reward than another trace whose run was very far. The computation of such rewards based on automata objectives were formalized in [22].

Here $\mathcal{A}_{\mathcal{R}}$ is used solely to compute rewards of traces, while the actual testing will be done by the MCTS algorithm. Let $C_0 = O$, and for $i \geq 1$, define $C_i = \text{Pre}_{\mathcal{A}_{\mathcal{R}}}(C_{i-1}) \setminus (C_0 \cup \dots \cup C_{i-1})$. We have $\bigcup_{i \geq 0} C_i = \text{CoReach}(\mathcal{A}_{\mathcal{R}}, O)$ and each C_i is the set of states that are at distance i from some state in O .

We consider two ways of assigning rewards to simulation traces. Let m be maximal such that $C_m \neq \emptyset$. Let us define $C_{m+1} = S_{\mathcal{A}_{\mathcal{R}}} \setminus \text{CoReach}(\mathcal{A}_{\mathcal{R}}, O)$, that is, all states that are not coreachable. We assign each trace σ whose run in $\mathcal{A}_{\mathcal{R}}$ ends in $s^{\mathcal{R}}$ the reward $\text{lastreward}(\sigma) = k$ if, and only if, $s^{\mathcal{R}} \in C_k$. Notice that this is well defined because the sets C_i are pairwise-disjoint and they cover all states. Hence, the closer the trace to objective O , the smaller its reward. A reward of 0 means that the state satisfies the objective.

The second reward assignment considers not only the last reward, but all rewards seen during the simulation, as follows. Let $r_0, r_1, r_2, \dots, r_{K-1}$ denote the sequence of rewards encountered during simulation (these are the rewards of the prefixes of the trace σ). If the length of the simulation was less than K , we simply repeat the last reward to extend this sequence to size K . Then the reward of the simulation is given by $\text{disc-reward}_{\gamma}(\sigma) = r_{K-1} \cdot \sum_{i=0}^{K-1} \gamma^i \cdot r_i$, where $\gamma \in (0, 1)$ is a discount factor. Notice that this value is 0, and minimal, if and only if $r_{K-1} = 0$. Furthermore, while r_{K-1} is the most important factor, the second factor means that we favor simulations whose first reward values are smaller. This can in fact be seen as a weighted version of *lastreward*, where the weight is smaller if the first steps have small rewards.

b) *Basic MCTS Testing Algorithm:* This yields the second testing algorithm we consider which we call *basic MCTS*. The algorithm is complete in the following sense since MCTS with UCT ensures that each node and action in the tree will be picked infinitely often in the limit.

Lemma 3.2: The basic MCTS algorithm is *complete*: For each requirement set $\mathcal{R}$, implementation I , test objective O , there exists $K > 0$ such that, if O is reachable in $\mathcal{A}_{\mathcal{R}} \otimes I$, then basic MCTS with simulation bound K finds a covering trace.

This algorithm is not just almost-surely complete, but also complete. This is because the UCT policy deterministically guarantees that all nodes of the tree are visited infinitely often. Thus, when the depth of the tree becomes large enough, any covering trace will be part of it, thus will have been executed. However, we do rely on estimated rewards to guide the search to ensure faster termination in practice. The above lemma holds

for both reward assignments *lastreward* and *disc-reward* _{γ} . Moreover, as for *uniform*, it is possible not to fix K , but increase it slowly towards infinity.

This basic MCTS algorithm is also a baseline since it can be obtained by combining known results from the literature following e.g. [13], [15].

B. Greedy Strategies and Improved Test Algorithms

We now describe our new test algorithms. We adopt a game-theoretic view of testing in the sense of games played on graphs [19], and define particular strategies for the tester, then show how these can improve the basic MCTS approach.

Finding a trace of $\mathcal{A}_{\mathcal{R}}$ that reaches O can be seen as a turn-based game played in the automaton $\mathcal{A}_{\mathcal{R}}$ between two players: the *tester*, and the *system*. At each step, the tester provides an input valuation, and the system responds with an output valuation, and the game moves to a new state in $\mathcal{A}_{\mathcal{R}}$. In this game, the online testing algorithm defines the strategy used by the tester, while the system plays a fixed strategy determined by the implementation, which is however black-box, thus unknown to the tester.

a) *Controllable Predecessor and Successor Operators:* Suppose we are given requirements specified as the complete deterministic safety automaton $\mathcal{A}_{\mathcal{R}} = \langle S_{\mathcal{A}_{\mathcal{R}}}, s_{\text{init}}^{\mathcal{A}_{\mathcal{R}}}, AP, T_{\mathcal{A}_{\mathcal{R}}}, F_{\mathcal{A}_{\mathcal{R}}} \rangle$ with $\text{Err}_{\mathcal{A}_{\mathcal{R}}} = S_{\mathcal{A}_{\mathcal{R}}} \setminus F_{\mathcal{A}_{\mathcal{R}}}$, and a test objective $O \subseteq S_{\mathcal{A}_{\mathcal{R}}}^{\text{in}}$. We consider the game $(\mathcal{A}_{\mathcal{R}}, O \cup \text{Err}_{\mathcal{A}_{\mathcal{R}}})$ played between the *tester* providing inputs, and the *system* providing outputs. The objective of the tester is to reach O or to reveal a non-conformance (that is, reach $\text{Err}_{\mathcal{A}_{\mathcal{R}}}$); it controls the input valuations in $2^{\text{AP}^{\text{in}}}$, and observes the output valuations in $2^{\text{AP}^{\text{out}}}$ chosen by the system.

A basic notion in the study of turn-based games is that of *controllable predecessors*. The *immediate controllable predecessors* of a set $B \subseteq S_{\mathcal{A}_{\mathcal{R}}}^{\text{in}}$ is the set $\text{CPre}_{\mathcal{A}_{\mathcal{R}}}(B) = \{s \in S_{\mathcal{A}_{\mathcal{R}}}^{\text{in}} \mid \exists \nu^{\text{in}} \in 2^{\text{AP}^{\text{in}}}, \forall \nu^{\text{out}} \in 2^{\text{AP}^{\text{out}}}, \text{Post}_{\mathcal{A}_{\mathcal{R}}}(s, \nu^{\text{in}} \cup \nu^{\text{out}}) \in B \cup \text{Err}_{\mathcal{A}_{\mathcal{R}}}\}$.

Thus, from each state $s \in \text{CPre}_{\mathcal{A}_{\mathcal{R}}}(B)$, the tester can select some valuation ν^{in} such that whatever the output ν^{out} returned by the system, we are guaranteed to either enter a state in B , or exhibit a non-conformance. We say that the tester *can ensure* entering B or reveal a non-conformance in one step (regardless of the system's strategy).

We define the set of *controllable predecessors* of a subset B as $\text{CPre}_{\mathcal{A}_{\mathcal{R}}}^*(B) = \text{lfp}(\lambda X. (B \cup \text{CPre}_{\mathcal{A}_{\mathcal{R}}}(X)))$. Using the same reasoning as previously, this least fixpoint defines the set of all states from which the tester can ensure either entering B within a finite number of steps, or reveal a non-conformance, regardless of the system's strategy.

For example, in Fig. 2, we have $\text{CPre}_{\mathcal{A}_{\mathcal{R}}}(O) = \{s_0, s'_0\}$ and $\text{CPre}_{\mathcal{A}_{\mathcal{R}}}^*(O) = \{s_0, s'_0\} \cup O$.

A *strategy* in this game is a function $f: \text{Tr}(\mathcal{A}_{\mathcal{R}}) \rightarrow 2^{2^{\text{AP}^{\text{in}}}}$ that associates each trace to a subset of input valuations, those that can be applied at the next step after this trace. A strategy f is said *memoryless* whenever, for any two traces σ and σ' reaching the same state in $\mathcal{A}_{\mathcal{R}}$, it holds $f(\sigma) = f(\sigma')$.

A trace $\sigma = \nu_1 \cdot \nu_2 \cdots \nu_n$ is *compatible* with a strategy f from a state s if there exists a sequence of states

$q_0, q_1, q_2, \dots, q_n$ such that $q_0 = s$ and for all $0 \leq i < n$, $q_i \xrightarrow{\nu_{i+1}^{\text{in}}} q_{i+1}$, and $\nu_{i+1}^{\text{in}} \in f(\nu_1 \dots \nu_i)$. We write $\text{Outcome}(f, s)$ to denote the set of all traces compatible with f from s , also called the *outcomes* of f from s .

b) *Winning and Cooperative Strategies*: For a given set $B \subseteq S^{\text{in}}$, a strategy f for the tester is *B-winning* from state s if all its outcomes from s eventually reach $B \cup \text{Err}_{\mathcal{A}_{\mathcal{R}}}$. A state s is *B-winning* if the tester has a winning strategy from s . The set of winning states is exactly $\text{CPre}_{\mathcal{A}_{\mathcal{R}}}^*(B)$:

Lemma 3.3: For all $B \subseteq S^{\text{in}}$, there exists a strategy f such that from all states $s \in \text{CPre}_{\mathcal{A}_{\mathcal{R}}}^*(B)$, all $\text{Outcome}(f, s)$ reach $B \cup \text{Err}_{\mathcal{A}_{\mathcal{R}}}$.

The strategy f provided by Lemma 3.3 is a *winning strategy* in the game $(\mathcal{A}_{\mathcal{R}}, B \cup \text{Err}_{\mathcal{A}_{\mathcal{R}}})$.

One could look for a winning strategy in $(\mathcal{A}_{\mathcal{R}}, O \cup \text{Err}_{\mathcal{A}_{\mathcal{R}}})$ in order to use it as a test algorithm. This approach was considered in some works (e.g., in [23]). However, in most cases, $\text{CPre}_{\mathcal{A}_{\mathcal{R}}}^*(O)$ does not contain the initial state; so such a strategy cannot be applied from the beginning. In terms of game theory, this means that the tester does not have a winning strategy from the initial state. We now discuss what to do if the initial state is outside $\text{CPre}_{\mathcal{A}_{\mathcal{R}}}^*(O)$.

c) *Cooperative Steps*: Given $B \subseteq S^{\text{in}}$, consider $\text{Pre}_{\mathcal{A}_{\mathcal{R}}}(B)$, the set of immediate predecessors of B : by definition, from these states, there exists a pair of valuations $(\nu^{\text{in}}, \nu^{\text{out}})$ for which the successor state is in B . From these states the tester cannot always guarantee reaching B in one step; however, it can choose an input valuation ν^{in} for which *there exists* ν^{out} which moves the system into B . In fact, when attempting to reach B , if the current state is not in $\text{CPre}_{\mathcal{A}_{\mathcal{R}}}^*(B)$, then choosing such a ν^{in} might be a good choice.

A strategy f is *B-cooperative* if for all $s \in \text{Pre}_{\mathcal{A}_{\mathcal{R}}}(B)$, and all $\nu^{\text{in}} \in f(s)$, there exists ν^{out} such that $\text{Post}_{\mathcal{A}_{\mathcal{R}}}(s, \nu^{\text{in}} \cup \nu^{\text{out}}) \in B$. As an example, in Fig. 2, in s_0^O , the O -cooperative strategy f is represented by a bold arrow. Choosing this input valuation *might* lead to O at the next step, while choosing the input valuation outside $f(s)$ surely leads outside O in the next step, for any output valuation.

In the rest of this section, we will combine winning and cooperative strategies to define *greedy* strategies, which can be seen as best-effort strategies that can be employed when there are no winning strategies from the initial state. These guarantee progress towards O in $\mathcal{A}_{\mathcal{R}}$ against *some* system strategies: because the implementation is arbitrary, the system strategy may not be “cooperative”, that is, at state $s \in \text{Pre}_{\mathcal{A}_{\mathcal{R}}}(B)$, given input $\nu^{\text{in}} \in f(s)$, it may not output ν^{out} ensuring $\text{Post}_{\mathcal{A}_{\mathcal{R}}}(s, \nu^{\text{in}} \cup \nu^{\text{out}}) \in B$. Nevertheless, we will show how to benefit from these strategies using randomization, using them as a heuristic to improve over the basic MCTS test algorithm.

d) *Greedy Strategy*: Consider $W_0 = \text{CPre}_{\mathcal{A}_{\mathcal{R}}}^*(O)$, which is non-empty since it contains O . Let f_0 be a O -winning strategy given by Lemma 3.3. If s_{init} belongs to W_0 , then f_0 ensures reaching $O \cup \text{Err}_{\mathcal{A}_{\mathcal{R}}}$ from s_{init} against all system strategies. In this case, we stop and return f_0 as the greedy strategy.

The interesting and more frequent case is when $s_{\text{init}} \notin W_0$. In this case, we inductively define an increasing sequence of

sets of states $W_0, W_1, \dots, W_n$ and corresponding strategies such that W_n is the set of all coreachable states.

Consider $i \geq 0$, and assume that the sequence has been defined until index i .

Let $\text{Coop}_{i+1} = \text{Pre}_{\mathcal{A}_{\mathcal{R}}}(W_i) \setminus W_i$ be the set of immediate predecessors of W_i deprived of states that have already been seen. Define f_{i+1}^c as a W_i -cooperative strategy. Then, let $W_{i+1} = \text{CPre}_{\mathcal{A}_{\mathcal{R}}}^*(\text{Coop}_{i+1} \cup W_i)$, and consider a corresponding winning strategy f_{i+1}^w . Let us denote by f_{i+1} as follows: for all $s \in S^{\text{in}}$, if $s \in \text{Coop}_{i+1}$, then $f_{i+1}(s) = f_{i+1}^c(s)$; if $s \in W_i$, then $f_{i+1}(s) = f_i(s)$; and if $s \in W_{i+1} \setminus (\text{Coop}_{i+1} \cup W_i)$, then $f_{i+1}(s) = f_{i+1}^w(s)$ (f_{i+1} is defined arbitrarily elsewhere).

We stop this sequence whenever Coop_i becomes empty which happens when $W_i = \text{CoReach}(\mathcal{A}_{\mathcal{R}}, O) = \text{Pre}_{\mathcal{A}_{\mathcal{R}}}^*(O)$.

The construction thus builds an increasing hierarchy $(W_i)_i$ of larger and larger sets, where at each level i , cooperation is needed in states of Coop_i to get closer to O in this hierarchy. For a state $s \in \text{CoReach}(\mathcal{A}_{\mathcal{R}}, O)$, we call the *rank* of s and denote $\text{rank}(s)$ the smallest index i such that $s \in W_i$. We denote by f_{greedy} the strategy defined by $f_{\text{greedy}}(s) = f_{\text{rank}(s)}(s)$ for all $s \in \text{CoReach}(\mathcal{A}_{\mathcal{R}}, O)$; and arbitrarily for other states. We call this the *greedy strategy*.

Notice that none of the winning strategies (f_i^w or f_0) composing f induces loops. However, since f is also composed of cooperative strategies, outcomes may contain loops due to outputs reaching states with higher ranks in the $(W_i)_i$ hierarchy. In other terms, f_{greedy} guarantees reaching $O \cup \text{Err}_{\mathcal{A}_{\mathcal{R}}}$ for some system strategies but not all; so it may induce infinite loops against some system strategies. Intuitively, the strategy f_{greedy} requires minimal cooperation from the system but it is also very optimistic.

The following example shows that the greedy strategy does not always guarantee progress towards the objective at each step. We will still use the greedy strategy in different ways to improve online testing and evaluate its effectiveness empirically.

Example 3.4: In Fig. 1, we have $W_0 = \{o\}$, $\text{Coop}_0 = \{s_1\}$, $\text{Coop}_1 = \{s_0\}$ while $f_{\text{greedy}}(s_0) = \{a\}$ and $f_{\text{greedy}}(s_1) = \{b, c\}$. The implementation given in Fig. 3 admits a covering trace which starts with $(b, 0)$. However, because $b \notin f_{\text{greedy}}(s_0)$, this cannot be found by the greedy strategy.

e) *Purely greedy and ϵ -greedy test algorithms*: The greedy strategy can be used to define a randomized test algorithm, called the *pure greedy* algorithm, and denoted **greedy**, as follows. Recall that after an active trace σ , the automaton $\mathcal{A}_{\mathcal{R}}$ is in a state $s_\sigma \in \text{CoReach}(\mathcal{A}_{\mathcal{R}}, O)$. We let **greedy** pick an input from $f(s_\sigma)$ uniformly at random. For $K > 0$, let **greedy** _{K} denote the test algorithm obtained from **greedy** by stopping each run after K steps, and restarting another run.

The algorithm **greedy** _{K} may not be almost-surely complete for any $K > 0$ (Example 3.4). However, we can obtain a almost-surely complete algorithm ϵ -**greedy** simply by randomizing between uniform and **greedy**: given $0 < \epsilon < 1$, this algorithm uses, at each state s_σ , **greedy** with probability $1 - \epsilon$; and uniform with probability ϵ . This ensures theoretical almost-sure completeness since if O is reachable in $\mathcal{A}_{\mathcal{R}} \otimes I$, then, at each run, there is a positive probability that the ϵ -**greedy** _{K}

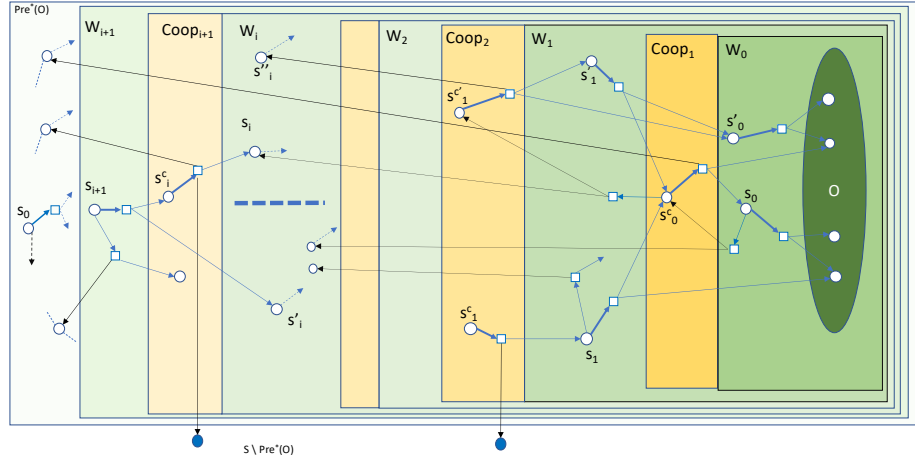


Fig. 2. Construction of the $(W_i)_i$ hierarchy and greedy strategies. States in S^{in} where the tester plays are represented by circles, while transient states in S^{out} where the system plays are represented by squares. In a state s_i in W_i , the strategy f_i is illustrated by bold blue arrows. Thin black arrows represent those outputs that either reach a state in a set W_k with higher rank or reach $S \setminus \text{CoReach}(\mathcal{A}_{\mathcal{R}}, O)$. For example, $s_1 \in W_1$ since the bold arrow ensures progressing to $\text{Coop}_1 \cup W_0$. However, $s_2 \in \text{Coop}_2$ since the bold arrow *can* make the game progress to W_1 but it can also go outside $\text{CoReach}(\mathcal{A}_{\mathcal{R}}, O)$ depending on the system strategy.

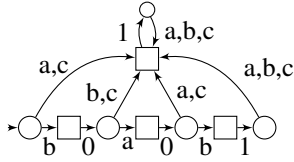


Fig. 3. An implementation with the only covering trace $(b, 0), (a, 0), (b, 1)$ for objective $\{o\}$ of the automaton of Fig. 1.

executes **uniform** for K steps, while picking the right inputs, and this will occur with probability 1 if these runs are repeated.

f) *Greedy-MCTS Test Algorithms:* Here, we explain our main contribution which is an improvement of the Basic MCTS test algorithm of Section III-A2 using the greedy strategy as heuristics both in the roll-out- and tree policies. Our heuristic accelerates convergence: it favors inputs that tend to get closer to the objective over those that do not, and if possible, more rarely use inputs that may lead to inconclusive states.

The first modification we make is using ϵ -**greedy** $_K$ as the roll-out policy instead of **uniform** $_K$, for some given K . We call this *MCTS+GRO* (for Greedy Rollout). The second modification consists in using the greedy policy within the tree policy. Fix a bound $M > 0$. During the expansion phase, in a given node whose projection in $\mathcal{A}_{\mathcal{R}}$ is s , we restrict the UCT policy to the inputs allowed by the greedy strategy $f_{\text{greedy}}(s)$ at the first M visits to that tree node, and then fallback to the regular UCT policy which covers the whole set of input valuations. We call this *MCTS+GTRO* (for Greedy Tree and Rollout). The bound M applies separately to each node, so at any moment, there are always nodes that have been explored less than M times at which the tree policy is restricted to the inputs of the greedy strategy.

Because the restriction to the greedy inputs only holds for a finite number of visits, this does not affect the convergence

guarantees of the MCTS algorithm. Our heuristic is intended to improve convergence by introducing a bias. Similar biased UCT scores have been used *e.g.* to solve the game Go [24].

IV. EXPERIMENTS

We implemented the test generation algorithm in Python where requirements are specified as deterministic automata in Verilog. These requirements are automatically translated to the AIGER format using Yosys and ABC, which is then read by the Absynthe game solver [25]. We implemented the computation of the sets $(W_i)_i$ and the greedy strategy of Section III-B within Absynthe. The testing tool uses the CUDD BDD library, and pyAIGER to read and execute the greedy strategy and the sets W_i computed by Absynthe, and to compute rewards. The tool and experiments are open-source and can be found at <https://github.com/osankur/rl-tester>¹.

We apply the presented testing algorithms on several instances of three systems: the flooding-time synchronization protocol (*FTSP*) with 3 and 4 processes [26], [27] where the test objective is to find an execution where all processes correctly elect a leader, a ring leader election protocol (*Ring*) with 16 and 32 processes [28], and a control program solving a planning problem (*Passageway*). We wrote our own implementations and requirement automata for these systems.

a) *Algorithms:* We compared all presented algorithms. These include the baseline algorithms **uniform** $_K$ and the basic MCTS, along with our algorithms ϵ -**greedy** $_K$ and MCTS with greedy roll-out (GRO) and MCTS with greedy tree policy and greedy roll out (GTRO).

We do not compare with [16] since the approach is not automatized and cannot consider automata-based requirements (but only user-provided waypoints). Further, note that in basic

¹The artifact for this paper can be downloaded from <https://doi.org/10.5281/zenodo.21507486>

TABLE I

AVERAGE EXECUTION TIMES IN SECONDS AND SUCCESS RATES (OVER 100 EXPERIMENTS). PERCENTAGES SHOW THE NUMBER OF SUCCESSFUL TRIALS OVER 100 EXPERIMENTS WITHIN THE TIME LIMIT (OTHERS STOPPED WITHOUT A RESULT AFTER THE TIME OUT). AN ABSENCE OF A PERCENTAGE IN A CELL MEANS 100% SUCCESS RATE. GIVEN DATA ARE FOR $\epsilon = 1/8$ (THE LEFT OF THE DOUBLE COLUMN), AND $\epsilon = 1/4$ (THE RIGHT OF THE DOUBLE COLUMN). NOTE THAT ϵ IS NOT RELEVANT FOR uniform_K AND NAIVE MCTS SO THESE ARE NOT REPRODUCED ON THE RIGHT. A DASH MEANS THAT NO TRACE WAS FOUND WITHIN THE TIME OUT. GRO STANDS FOR GREEDY ROLL-OUT, AND GTRO FOR GREEDY TREE POLICY AND ROLL OUT. THE BOLD ENTRIES SHOW THE BEST PERFORMING ALGORITHM(S) ON EACH ROW AMONG ALGORITHMS WITH 100% SUCCESS RATE.

	uniform_K	$\epsilon\text{-greedy}_K$	MCTS	MCTS + GRO	MCTS + GTRO	$\epsilon\text{-greedy}_K$	MCTS + GRO	MCTS + GTRO
FTSP 3	-	226.9 (67%)	275.6 (88%)	138.7	77.6	210.6	178.3	131.8
FTSP 4	-	-	407.0 (23%)	456.0	290.7	-	438.9	341.9
Ring 16	-	0.6	96.3	0.7	0.7	0.9	0.4	0.9
Ring 32	-	1.1	216.3 (94%)	1.1	1.2	56.4	59.4	16.3
Passage 3	113.2 (98%)	6.0	7.3	1.8	0.5	6.7	2.0	1.2
Passage 5	-	169.7 (66%)	47.1	17.8	2.8	218.2 (41%)	20.7	2.8
Passage 10	-	-	429 (3%)	300 (9%)	15.0	-	327 (14%)	15.2
FTSP' 3	72.3	51.8	64.2	44.2	41.0	47.9	36.9	47.4
FTSP' 4	126.7 (3%)	105.5 (3%)	468.5 (34%)	419.6 (34%)	438.6 (25%)	85 (2%)	434.9 (36%)	459.3 (30%)
FTSP'' 3	-	217.0 (68%)	303.2 (69%)	179.7 (98%)	110.6	222.6 (11%)	220.4 (95%)	154.5 (98%)
FTSP'' 4	-	-	416.6 (8%)	498.8 (2%)	364.5 (77%)	-	427.7 (15%)	356.6 (73%)

MCTS, we use rewards that are based on remaining distances in our hierarchy. This can be seen as a direct application of [22] to model-based testing, where a reinforcement learning was applied with a reward function based on the distance to the goal. In fact, we see the basic MCTS as a representative of the approach of using an RL-based algorithm with a clever choice of rewards. Last, we also do not compare with [15] whose reward functions are highly dependent on LTL formulas; but their RL-based approach can be seen as being represented by our basic MCTS.

We initially used Torxakis as a baseline but do not present these results here. The reason is that it failed to reach the test objective even for small instances so the performance was systematically below our implementation of uniform_K . We believe that this might be because Torxakis does not analyze the state space of the requirements automaton so it does not eliminate inputs that lead to inconclusive states. Furthermore, the input language is very complex and it was difficult to interpret the error messages of the compiler, so we only made the modeling effort of Passageway.

b) Parameters: Each algorithm made up to 50,000 runs where each run generates a trace of length $K = 1000$, with an overall time-out of 600s. We repeated each experiment 100 times in order to compute average running times. For example, for the basic MCTS, we started from scratch 100 times, and built the tree up to 50,000 nodes, making a roll-out of length 1000 for each added node.

In these experiments, we set $\epsilon = 1/8$ and $M = 1000$ which were empirically determined to give best results. In fact, we also experimented with lower values of M (from 0 to 1000) and the highest value was best in our cases, and it was not useful to increase M any further. In our experiments, low values of ϵ performed better, while decreasing ϵ any further did not considerably change the outcome. The choice of $K = 1000$ is somewhat arbitrary. The results were not impacted by choosing larger values (while choosing significantly smaller values meant that objectives could never be reached).

We report here the running times and success rates once these parameters were fixed. As in many applications of reinforcement learning, there is no obvious *a priori* choice of

the parameters, and finding good parameters is often part of the process. In practice, K can be chosen if one has a good guess for an upper bound on the length of executions. As for the choice of ϵ and M , one could start with small values for ϵ and large values of M which will strongly favor greedy strategies, and decrease gradually in case this does not work. In our results, we show execution times for $\epsilon = \frac{1}{4}$ as well to assess the sensitivity of the performance on the choice of ϵ .

c) Results: The results of various algorithms on the described systems are given in Table I. We comment here on the FTSP benchmarks. FTSP is a leader election algorithm where processes run in quasi-synchrony (they are asynchronous but in a bounded way - see [27]), can crash, and recover. The tester selects at each step the process to execute, and whether the selected process must crash. The requirements for FTSP ensures this property of the scheduling except that a process recovering from a fault can execute more often. Moreover, the automaton tracks which process has correctly elected a leader, and which one is ready to broadcast the leader information (which will in turn ensure other processes elect the leader). In FTSP, along a normal execution each process can be in three phases: A) initially the process has no leader (or has the wrong leader), B) eventually receives a message about the correct leader, and C) eventually gathers enough messages from the leader so that it starts broadcasting itself. Modelling these three phases for each process in the requirements helped the test algorithm guide the executions towards the objective.

As most benchmarks, FTSP was too hard for uniform_K , but introducing learning with MCTS or using greedy strategies were successful: more precisely, on FTSP 3, basic MCTS and $\epsilon\text{-greedy}_K$ both succeeded but with less than 100% rate. A success rate of 100% is achieved by combining both while MCTS + GTRO was the fastest. The same success rate but higher execution times were achieved with $\epsilon = 1/4$. The model FTSP 4 is larger which can be seen on the lower success rates and higher execution times.

We also experimented with two variants of the FTSP benchmark (FTSP' and FTSP'') where the implementation was identical but the requirements automata were different. In FTSP', the requirements strictly ensure the quasi-synchrony

condition regardless of the recovery phase. This meant that inputs violating this condition now necessarily lead outside of $\text{CoReach}(\mathcal{A}_R, O)$, and are thus removed from the set of possible inputs for all algorithms. Inconclusive states were therefore no longer reachable. This is why uniform_K performs relatively well in these cases.

FTSP'' is an example showing the limits of the approach and the importance of selecting a detailed model. Here, the implementation is the same as FTSP, but the requirements only track the scheduling constraint and whether each process has elected a leader, but not whether a process is ready to broadcast or not (phase C above). Thus rewards computed from the requirements carry less information, which explains lower success rates. In particular basic MCTS and MCTS+GRO rarely find solutions for FTSP'' 4, while MCTS+GTRO achieves success rate of 77%: compare this with 100% success rate for FTSP 4. This is consistent with [16] where handcrafted waypoints (intermediate objectives) were necessary to achieve coverage and good convergence time. In our cases, these "waypoints" correspond to the states of the requirement automaton.

The advantage of the greedy strategies can be seen clearly in the ring protocol. Here the tester selects at each step the type of crash that should occur in the system, so the greedy strategies are useful to avoid selecting crashes that delay reaching leader election. In this case, learning had some advantage, especially for $\epsilon = 1/4$, but we believe that the success is mainly due to the greedy strategy which focuses on useful inputs.

The preprocessing times (time required to solve the game, and build the hierarchy) were below 1s in each case with the following exceptions: 2.5s for FTSP4' and 31s for FTSP4 (with some variability due to randomization in the BDD package). The largest BDD sizes observed were for the FTSP4 benchmarks: for FTSP4, FTSP4', FTSP4'', we measured, respectively, 74927, 9231, 468 nodes.

All requirement automata are encoded as Verilog modules so we report below the number of registers (bits) used in these modules. This gives an upper bound on the size of the automata (for n bits, there can be up to 2^n states) although the reachable part of the state space might be smaller.

FTSP3 and FTSP3'	38 bits
FTSP3''	28 bits
FTSP4 and FTSP'	46 bits
FTSP4''	36 bits
Ring	14 bits
Passageway	10 bits

Peak memory consumption (RSS) measured with gnu time was 57MB for Ring32, 750MB for FTSP4, and 57MB for Passageway10.

V. MORE ON RELATED WORKS

The notion of cooperative states have been used in [29] in the context of offline test synthesis from timed automata, inspired by the previous approach of [30]. In [31], in the context of timed systems, the authors rely on cooperative strategies to synthesize test cases when the cooperation of the system is required for winning. However, these yield incomplete testing methods ,

or completeness is obtained by making strong assumptions on the implementations; the novelty of our approach is to obtain a complete method by using these notions in reinforcement learning. A discussion on model-based testing techniques can be found in [32].

Several test generation tools based on the **ioco** conformance relation for input-output labelled transition systems (IOLTS) have been developed. Roughly, an implementation *ioco*-conforms to its specification if after any of its observed behaviour that is also a specification behaviour, the implementation only produces outputs or quiescences that are also possible in the specification. The tools TGV [20] and TESTOR [33] generate off-line test cases from formal specifications in various languages with IOLTS semantics, driven by test purposes described by automata. The tools JTorx [34] and TorXakis [9] are improvement of TorX [8] and allow to generate and execute online test cases derived from various specification languages. In the context of timed models, Uppaal-TRON [35] is an online test generation tool for timed automata based on the real-time extension **rtioco** of the **ioco** conformance relation.

In [16], the authors develop a testing framework based on Q-learning with dynamic rewards that encourage a good coverage of a system, and apply their methods to test distributed algorithms. In particular, the Cube World example they develop is similar in spirit to our passageway system. Moreover, they guide the reinforcement learning agent using *waypoints* which are important steps that mark progress towards the objective in the analyzed system. Our automata-based framework can be seen as a generalization of waypoints since requirements can express sequences of particular events in the system, but also more complex behaviors.

[36] uses Q-learning to produce tests for GUI applications but without a model for the specification: the objective is to reach a large number of visually different states. RL-based testing for GUI has attracted significant attention. [37] uses Q-learning with the aim of covering as many states as possible; see also [38]. In [39], reinforcement learning is used to compute *valid* inputs for testing programs: these consist in producing inputs that satisfy the precondition of a program to be tested so that assertions can be checked. [40] uses RL to learn short synchronizing sequences, where rewards correspond to the size of the powerset of states. In [41], reinforcement learning was used to test shared memory programs. Deep reinforcement learning has also been used for Android testing; see e.g. [42]. [43] combines blackbox testing and model learning in order to improve coverage.

Reward shaping for automata-based specifications has been considered for Monte Carlo Tree Search. In [22], the approach is based on the distance to accepting states of Büchi automata; and in [44], the authors collect statistics on the success for all transitions on the specification automaton. The latter approach is not adapted to our case, where the goal is to find a single successful execution, and not to actually learn the optimal values at all states.

VI. CONCLUSION

While the game point of view has been explored before in requirement-based testing, we used it to improve the reinforcement learning approach. Our preliminary experiments show that such heuristics can improve the testing time by guiding the search quickly to relevant states, especially when test objectives require long sequences that have low probability to be guessed randomly.

Our method applies to systems under test interacting with an environment by alternating inputs and outputs, so the system must be instrumented to interact with the test driver. The method moreover needs requirement automata to be provided, which are not always obvious to build. While we wrote requirements of our benchmarks directly as automata, one might also obtain automata from requirements written in structured natural language [45] or dedicated languages based on temporal logic [46], [47]. Moreover, there is a large body of work on the TLA+ formalization of distributed algorithms (e.g. [48]), and many works report the benefits of testing w.r.t. such formal requirements [49]. We will investigate the extension of our approach to such specifications in future work. The definition of games on such formalisms and extending game solvers to such languages are difficulties that need to be addressed.

Our approach consists in testing behaviors against temporal, i.e. multi-steps, specifications, which means functional testing of complex behaviors of a system under test (SUT). Our approach is insensitive to choices in the implementation, so it is suitable when SUT is changed, and can efficiently find relevant test cases (i.e. each test case being a list of successive input values) without user intervention.

We considered here system-level testing where requirements refer to the behaviors of the whole system. One might also consider this approach for unit-testing components. In this case, the requirement automaton would refer to the behaviors of the component which is isolated possibly using stubs following standard unit-testing practices.

DATA AVAILABILITY

The artifact(s) associated with this paper is/are available at Zenodo under the following DOI: 10.5281/zenodo.21507486.

APPENDIX

A. Omitted Definitions

Definition A.1: Given automata $\mathcal{A}_1 = \langle S_1^{\text{in}} \uplus S_1^{\text{out}}, s_{\text{init},1}, AP_1, T_1, F_1 \rangle$ and $\mathcal{A}_2 = \langle S_2^{\text{in}} \uplus S_2^{\text{out}}, s_{\text{init},2}, AP_2, T_2, F_2 \rangle$, their product $\mathcal{A}_1 \otimes \mathcal{A}_2$ is an automaton $\mathcal{A} = \langle S, s_{\text{init}}, AP, T, F \rangle$ where $S = (S_1^{\text{in}} \times S_2^{\text{in}}) \uplus (S_1^{\text{out}} \times S_2^{\text{out}})$, $s_{\text{init}} = (s_{\text{init},1}, s_{\text{init},2})$, $AP = AP_1 \cup AP_2$, $F = F_1 \times F_2$ and the set of transitions is defined as follows: there is a transition $((s_1, s_2), g, (s'_1, s'_2))$ in T if there are transitions (s_1, g_1, s'_1) in T_1 and (s_2, g_2, s'_2) in T_2 with $g = g_1 \wedge g_2$.

Notice that this definition indeed yields an automaton in the sense of Def. 2.1 and preserves completeness and determinism. Moreover, the product of two safety automata is a safety automaton: the set of accepting states is $F = F_1 \times F_2$, so the set **Err** in the product automaton is $(\mathbf{Err}_1 \times S_2^{\text{in}}) \uplus$

$(S_1^{\text{in}} \times \mathbf{Err}_2)$, and thus inherits absorbance. The product of automata is commutative and associative, and can thus be generalized to an arbitrary number of automata.

B. Additional Examples

The next example shows how an automaton can be obtained from requirements for a simple factory automation system.

Example A.2 (Factory Carriage Example): We consider a controller program in a factory automation system depicted in Fig. 4. In this system, when the carriage is on the right end (bwdlimit) and receives a cargo on top of it, the controller program must move the carriage forward (movefwd) until it reaches the forward limit (fwdlimit). The controller must then push the arm for 3 seconds, and it can only back the carriage up (movebwd) after this duration.

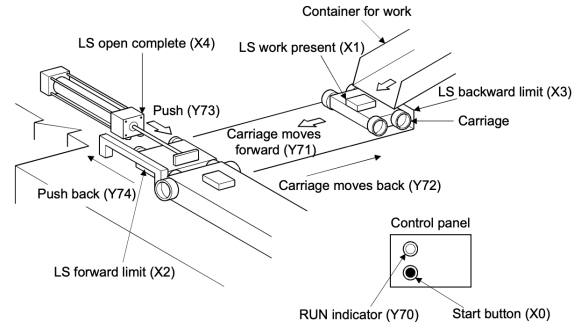


Fig. 4. The carriage control system example from [50].

For the sake of this example, we only model the requirements concerning the first phase, that is, until the carriage reaches the forward limit. Three of these requirements on the controller program are given below.

- R1 When the carriage is on its backward limit, and a cargo is present, then it immediately moves forwards until reaching the forward limit.
- R2 If the carriage is not already moving forward, and if no cargo is present or the carriage is not in the backward limit, then it is an error to move forward. The carriage must never be moved forward and backwards at the same time.
- R3 When some cargo is present, and the carriage is at its forward limit, it should stop moving forward immediately.

All these three requirements are modelled in the automaton of Fig. 5. The initial state is s_0 , and we distinguish the state **err** which makes this a safety automaton. Intuitively, state s_2 is reached when the carriage receives a cargo and brings it successfully to the forward limit.

The following example illustrates the product of an implementation with its requirement automaton.

Example A.3: We consider the requirement expressed by the automaton of Fig. 1, the objective defined by the singleton $O = \{o\}$, and the implementation represented to the left of Fig. 6. Their product is represented in the right of Fig. 6. There is a covering trace in this case since the input sequence ab generates $(a0b1)$ in I_1 , reaching o in $\mathcal{A}_R$.

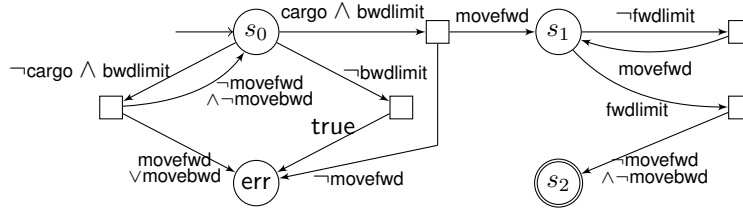


Fig. 5. An automaton modeling requirements R1, R2, R3. For readability, we omitted some transitions in the figure: from all output states, there is an additional transition guarded by $\text{movebwd} \wedge \text{movefwd}$ to **err**. Moreover, from each of the two rightmost output states, the negation of the guard of the only leaving transition goes to **err** as well. For instance, from s_1 , reading $\neg \text{fwdlimit}$ and then $\neg \text{movefwd}$, we end in **err**. Note that reading $\neg \text{bwdlimit}$ in s_0 is also an error since this is not supposed to happen in this system.

Assume now that I_1 is modified so that it outputs 1 on input a from s_0^I , then the product would go to a state of the form (t, s_0^I) . If t is an **Err** state, then the implementation does not conform to the requirement; if not, then the test is inconclusive since o is no longer reachable.

On the other hand, consider an implementation outputs 0 on any input. Then any trace is an active trace although the implementation does not have a covering trace (in fact, the product cannot reach a state involving o).

C. Omitted Proofs

Proof of Lemma 3.1. Assume there exists a covering trace σ in $\mathcal{A}_{\mathcal{R}} \otimes I$, and let K be the length of σ . When playing uniform_K ad infinitum, the testing process restarts an infinite number of times. At each step, the algorithm picks each valuation of the input variables with probability $2^{-|\text{AP}^{\text{in}}|}$. So at each restart, the probability of choosing exactly σ is $2^{-|\text{AP}^{\text{in}}| \cdot |\sigma|}$. Therefore, σ is picked eventually with probability 1. $\square$

Proof of Lemma 3.3. Consider the computation of

$$\text{CPre}_{\mathcal{A}_{\mathcal{R}}}^*(B) = \text{lfp}(\lambda X. (B \cup \text{CPre}_{\mathcal{A}_{\mathcal{R}}}(X))),$$

and let $C_0, C_1, \dots, C_n$ denote the iterates such that $C_0 = \emptyset$, and $C_i = B \cup C_{i-1} \cup \text{CPre}(C_{i-1})$ for $i \geq 1$. For each state $s \in \text{CPre}_{\mathcal{A}_{\mathcal{R}}}^*(B)$, let i_s denote the least index such that $s \in C_{i_s}(B)$. We define $f(s) = \{\nu^{\text{in}} \mid \forall \nu^{\text{out}}, \text{Post}_{\mathcal{A}_{\mathcal{R}}}(s, \nu^{\text{in}} \cup \nu^{\text{out}}) \in C_{i_s-1} \cup \text{Err}_{\mathcal{A}_{\mathcal{R}}}\}$. $\square$

D. Non-Deterministic Implementations

Although we restricted the presentation of the above algorithms to deterministic implementations, they all apply to non-deterministic ones (that is, automata that are neither input-deterministic nor output-deterministic). If the implementation is finite-state and purely randomized, that is, if it can be modelled by a finite-state Markov chain, then all algorithms apply with the same completeness guarantees since MCTS and similar algorithms have been designed for Markov decision processes. On the other hand, if non-deterministic choices do not follow probability distributions, or cannot be modelled by a finite-state Markov chain, then the algorithms do not have completeness guarantees. This is typically the case if the implementation is initialized in arbitrary state (e.g., due to states of the caches) and possible initializations do not follow any particular probability distribution. As usual in reinforcement learning, MCTS can

still be applied and might converge or give useful results even though no theoretical guarantees can be proven. Thus, all our algorithms can be applied in practice to non-deterministic implementations. We leave an empirical evaluation in such cases for future work.

REFERENCES

- [1] M. R. Blackburn and R. D. Busser, “T-VEC: A tool for developing critical systems,” in *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS’96*. IEEE, 1996, pp. 237–249.
- [2] J. Tretmans and P. van de Laar, “Model-based testing with TorXakis,” in *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin, 2019, pp. 247–258.
- [3] B. Jeannot and F. Gaucher, “Debugging embedded systems requirements with STIMULUS: an automotive case-study,” in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.
- [4] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Software testing, verification and reliability*, vol. 22, no. 5, pp. 297–312, 2012.
- [5] W. Li, F. Le Gall, and N. Spaseski, “A survey on model-based testing tools for test case generation,” in *Tools and Methods of Program Analysis: 4th International Conference, Tmpa 2017, Moscow, Russia, March 3-4, 2017, Revised Selected Papers 4*. Springer, 2018, pp. 77–89.
- [6] W. Bolton, *Programmable logic controllers*. Newnes, 2015.
- [7] B. Beizer, *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.
- [8] G. Tretmans and H. Brinksma, “Torx: Automated model-based testing,” in *First European Conference on Model-Driven Software Engineering*, 2003, pp. 31–43.
- [9] “Torxakis,” <https://github.com/torxakis>.
- [10] J. Tretmans, “Test generation with inputs, outputs and repetitive quiescence,” *Softw. Concepts Tools*, vol. 17, no. 3, pp. 103–120, 1996.
- [11] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
- [12] M. Yannakakis, “Testing, optimization, and games,” in *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, 2004, pp. 78–88.
- [13] M. Veanes, P. Roy, and C. Campbell, “Online testing with reinforcement learning,” in *Formal Approaches to Software Testing and Runtime Verification*, K. Havelund, M. Núñez, G. Roşu, and B. Wolff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 240–253.
- [14] A. Y. Ng, D. Harada, and S. Russell, “Policy invariance under reward transformations: Theory and application to reward shaping,” in *ICML*, vol. 99. Citeseer, 1999, pp. 278–287.
- [15] Y. K roglu and A. Sen, “Functional test generation from UI test scenarios using reinforcement learning for testing distributed systems,” *Softw. Test. Verification Reliab.*, vol. 31, no. 3, 2021. [Online]. Available: <https://doi.org/10.1002/stvr.1752>
- [16] A. Borgarelli, C. Enea, R. Majumdar, and S. Nagendra, “Reward augmentation in reinforcement learning for testing distributed systems,” *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. [Online]. Available: <https://doi.org/10.1145/3689779>
- [17] R. Coulom, “Efficient selectivity and backup operators in monte-carlo tree search,” in *International conference on computers and games*. Springer, 2006, pp. 72–83.

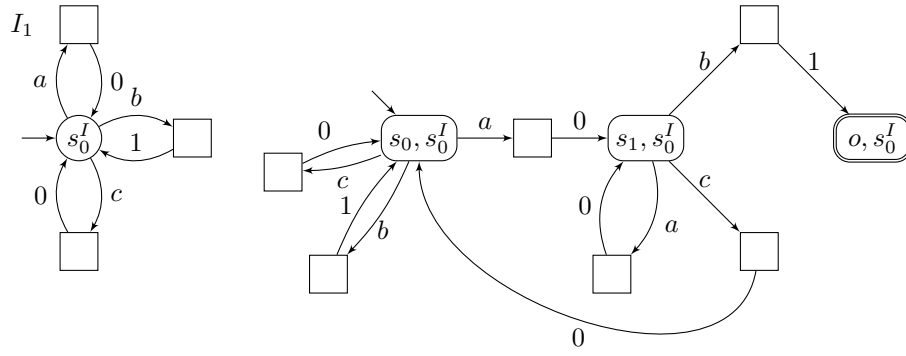


Fig. 6. An implementation I_1 and its product with the automaton of Fig. 1

- [18] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.
- [19] N. Fijalkow, C. Aiswarya, G. Avni, N. Bertrand, P. Bouyer, R. Brenguier, A. Carayol, A. Casares, J. Fearnley, H. Gimbert, T. A. Henzinger, P. Gastin, F. Horn, R. Ibsen-Jensen, N. Markey, B. Monmege, P. Novotný, P. Ohlmann, M. Randour, O. Sankur, S. Schmitz, O. Serre, M. Skomra, N. Sznajder, and P. Vandenhover, *Games on Graphs: From Logic and Automata to Algorithms*. Cambridge University Press, 2025, vol. abs/2305.10546, in press, Cambridge University Press. [Online]. Available: <https://doi.org/10.48550/arXiv.2305.10546>
- [20] C. Jard and T. Jérón, "TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *International Journal on Software Tools for Technology Transfer*, vol. 7, pp. 297–315, 2005.
- [21] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings 17*. Springer, 2006, pp. 282–293.
- [22] A. Camacho, R. T. Icarte, T. Q. Klassen, R. A. Valenzano, and S. A. McIlraith, "LTL and beyond: Formal languages for reward function specification in reinforcement learning," in *IJCAI*, vol. 19, 2019, pp. 6065–6073.
- [23] A. David, K. G. Larsen, S. Li, and B. Nielsen, "A game-theoretic approach to real-time system testing," in *Proceedings of the 2000 Design, Automation and Test in Europe (DATE'00)*. IEEE Comp. Soc. Press, Mar. 2008, pp. 486–491.
- [24] S. Gelly and D. Silver, "Monte-carlo tree search and rapid action value estimation in computer go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.
- [25] R. Brenguier, G. A. Pérez, J. Raskin, and O. Sankur, "Absynthe: abstract synthesis from succinct safety specifications," in *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014*, ser. EPTCS, K. Chatterjee, R. Ehlers, and S. Jha, Eds., vol. 157, 2014, pp. 100–116. [Online]. Available: <https://doi.org/10.4204/EPTCS.157.11>
- [26] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi, "The flooding time synchronization protocol," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004, pp. 39–49.
- [27] O. Sankur and J.-P. Talpin, "An abstraction technique for parameterized model checking of leader election protocols: Application to ftsp," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 23–40.
- [28] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes," *Communications of the ACM*, vol. 22, no. 5, pp. 281–283, 1979.
- [29] L. Henry, T. Jérón, and N. Markey, "Control strategies for off-line testing of timed systems," in *Model Checking Software - 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings*, ser. Lecture Notes in Computer Science, M. Gallardo and P. Merino, Eds., vol. 10869. Springer, 2018, pp. 171–189. [Online]. Available: https://doi.org/10.1007/978-3-319-94111-0_10
- [30] S. Ramangalahy, "Strategies for conformance testing," Max Planck Institut Für Informatik, Tech. Rep. MPI-I-98-010, May 1998. [Online]. Available: <https://hdl.handle.net/21.11116/0000-0001-7559-8>
- [31] A. David, K. Larsen, S. Li, and B. Nielsen, "Cooperative testing of timed systems," in *4th Workshop on Model Based Testing (MBT'08)*, vol. 220, 12 2008, pp. 79–92.
- [32] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with Spec Explorer," in *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 39–76. [Online]. Available: https://doi.org/10.1007/978-3-540-78917-8_2
- [33] L. Marsso, R. Mateescu, and W. Serwe, "Testor: A modular tool for on-the-fly conformance test case generation," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 211–228.
- [34] A. Belinfante, "Jtorx: A tool for on-line model-driven test derivation and execution," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2010, pp. 266–270.
- [35] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using uppaal," *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, pp. 77–117, 2008.
- [36] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Autoblacktest: A tool for automatic black-box testing," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1013–1015. [Online]. Available: <https://doi.org/10.1145/1985793.1985979>
- [37] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android GUI testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating Test Case Design, Selection, and Evaluation*, 2018, pp. 2–8.
- [38] Z. Lv, C. Peng, Z. Zhang, T. Su, K. Liu, and P. Yang, "Fastbot2: Reusable automated model-based GUI testing for android enhanced by reinforcement learning," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3551349.3559505>
- [39] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1410–1421.
- [40] U. C. Türker, R. M. Hierons, M. R. Mousavi, and I. Y. Tyukin, "Efficient state synchronisation in model-based testing through reinforcement learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 368–380.
- [41] N. Pfeifer, B. V. Zimpel, G. A. G. Andrade, and L. C. V. dos Santos, "A reinforcement learning approach to directed test generation for shared memory verification," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 538–543.
- [42] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement

- learning for black-box testing of android apps,” *ACM Trans. Softw. Eng. Methodol.*, 2021. [Online]. Available: <https://doi.org/10.1145/3502868>
- [43] R. Fogler, I. Cohen, and D. Peled, “Accelerating black box testing with light-weight learning,” in *Model Checking Software*, G. Caltais and C. Schilling, Eds. Cham: Springer Nature Switzerland, 2023, pp. 103–120.
 - [44] A. Velasquez, B. Bissey, L. Barak, A. Beckus, I. Alkhouri, D. Melcer, and G. Atia, “Dynamic automaton-guided reward shaping for Monte Carlo tree search,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 13, pp. 12 015–12 023, 2021. [Online]. Available: <https://ojs.aaai.org/index.php/AAAI/article/view/17427>
 - [45] D. Giannakopoulou, A. Mavridou, J. Rhein, T. Pressburger, J. Schumann, and N. Shi, “Formal requirements elicitation with fret,” in *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020)*, no. ARC-E-DAA-TN77785, 2020.
 - [46] S. Becker, D. Dietsch, N. Hauff, E. Henkel, V. Langenfeld, A. Podelski, and B. Westphal, “Hanfor: Semantic requirements review at scale,” in *REFSQ Workshops*, ser. CEUR Workshop Proceedings, vol. 2857. CEUR-WS.org, 2021.
 - [47] O. Sankur, R. Noguchi, T. Jérón, and D. Mentré, “Reqkit: A real-time requirement analysis tool kit,” Tech. Rep., 2025.
 - [48] N. Bertrand, P. Ghorpade, S. Rubin, B. Scholz, and P. Subotić, “Reusable formal verification of dag-based consensus protocols,” in *NASA Formal Methods*, A. Dutle, L. Humphrey, and L. Titolo, Eds. Cham: Springer Nature Switzerland, 2025, pp. 138–158.
 - [49] A. J. J. Davis, M. Hirschhorn, and J. Schvimer, “Extreme modelling in practice,” *Proc. VLDB Endow.*, vol. 13, no. 9, p. 1346–1358, May 2020. [Online]. Available: <https://doi.org/10.14778/3397230.3397233>
 - [50] Mitsubishi Electric Corporation, “Mitsubishi programmable controller – Training manual,” 2012, https://dl.mitsubishielectric.com/dl/fa/document/manual/school_text/sh081123eng/sh081123enga.pdf.