# Online Test Synthesis From Requirements: Enhancing Reinforcement Learning with Game Theory

Ocan Sankur, Thierry Jéron, Nicolas Markey, David Mentré, Reiya Noguchi

# Online Test Synthesis From Requirements: Enhancing Reinforcement Learning with Game Theory

Ocan Sankur[1], Thierry Jéron[1], Nicolas Markey[1], David Mentré[2], and Reiya Noguchi[3]

[1]Univ Rennes, Inria, CNRS, Rennes, France, firstname.lastname@inria.fr
[2]Mitsubishi Electrics R&D Centre Europe, Rennes, France,
initial-of-firstname.lastname@fr.merce.mee.com
[3]Mitsubishi Electric Corporation, Tokyo, Japan,
lastname.firstname@ah.MitsubishiElectric.co.jp

## 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, so as to satisfy a coverage criterion, while monitoring the violation of the requirements. We develop an approach based on Monte Carlo Tree Search, which is a classical technique in reinforcement learning for efficiently selecting promising inputs. 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 the Monte Carlo Tree Search algorithm, thus improving the performance of testing.

## 1 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 [BB96], TorXakis [TvdL19] or Stimulus [JG16], and surveys [UPL12] and [LLGS18]).This helps the development process since the developer can focus on writing formal specifications or requirements, and test cases are generated automatically. There are two main approaches for test generation: *black-box testing* focuses on generating tests without having access to the system's internals such as its source code (see e.g., [Bei95]), while *white-box testing* generates tests by analyzing its source code (see e.g., [MBTS04, AO17]).

**Black-Box Testing.** In this paper, we are interested in automatically synthesizing online test cases from a set of requirements and an implementation under test in a black-box setting. By *black-box*, we mean that the implementation internal is unknown to the tester (or at least not used), only its interaction with her is used. By *online*, we mean that the synthesis is essentially performed during execution, while interacting with the implementation. We consider implementations that

are reactive: these are programs that alternate between reading an input valuation and writing an output valuation. This setting is interesting for modeling synchronous systems, e.g., controllers for manufacturing systems [Bol15]. The considered requirements are given as automata recognizing 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 requirements automaton.

The goal of the test cases is to drive the implementation to some particular state or show some particular behaviour where non-conformance is suspected to occur. These are described by *test objectives*. They are typically derived from coverage criteria, e.g. state or transition coverage, or written from requirements []. The selection of the test objectives is out of the scope of this paper; we thus assume that these are given.

There are several black-box testing algorithms and tools in the literature. The closest to our approach is TorXakis [TvdL19, Tor] which is a tool based on the **ioco** testing theory [Tre96] and the previous TorX tool [TB03]. It allows the user to specify the automata-based requirements reading input-output valuations as well as test objectives (called test purposes) in a language based on process algebra, and is able to generate online tests interacting with a given implementation. These tests are performed 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.

**Reinforcement Learning for Testing.** This issue has been addressed in many works by interpreting the test synthesis in a *reinforcement learning*(RL) setting [SB18]. 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. It has been applied to learn strategies for playing board games such as Chess and Go [SHM$^+$16]. Here the test synthesis 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 [Yan04]; and online testing for interface automata specifications were considered in [VRC06] 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* [NHR99] 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 [KS21] 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.

**Contributions.** Although reinforcement learning helps one to guide the search towards the test objective, these methods can still be slow in finding traces satisfying the test objective, especially when the number of input bits is high, and when the traces to be found are long.

In this work, we target improving the performance of black-box online test algorithms 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 [Cou06] (see also e.g., the survey [BPW$^+$12]) is a RL technique to search

for good moves 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, MCTS builds a weighted tree of possible plays of the game, while the decision of which branch to explore at each step is based on a random selection appropriately biased to select unexplored moves but also moves with high reward estimates. The *tree policy* is the policy used for exploring the branches of the constructed tree, while the *roll-out policy* is used to run a long execution to estimate the overall reward.

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. The heuristic is based on a *greedy* test strategy computed as follows. We adopt the game-theoretic view and see the testing process as a game played on the requirement automaton state space. At any step, when the tester provides an input, we assume that the implementation can also answer with any output. This defines a zero-sum game: the tester has the objective of reaching the test objective, and the implementation has the objective of avoiding it. We first consider *winning* strategies in this game: if there is a strategy for the tester which prescribes inputs such that the test objective is reached no matter what the implementation outputs, then this strategy is guaranteed to reach the test objective. However, in general, there are no winning strategies from all states. In this case, we compute winning strategies for the tester to reach so-called *cooperative* states, from where *some* output that the implementation can provide reduces the distance to the test objective in the requirement automaton. This is an optimistic strategy: if the implementation "cooperates", that is, provides the right outputs at cooperative states, this guarantees the reachability of the test objective; but otherwise, no guarantee is given. The greedy strategy consists in selecting uniformly at random inputs that are part of a winning strategy if any, or allow the implementation to cooperate.

Our variant of the MCTS algorithm uses the standard UCT algorithm [KS06] as a tree policy, but restricted to those inputs that are part of the greedy strategy at the first $M$ visits at each node of the tree; after the $M$-th visit to a node, the UCT policy is applied to the set of all inputs. Moreover, the roll-out policies use $\epsilon$-greedy strategies, which consists in selecting inputs uniformly at random with probability $\epsilon$, and using the greedy strategy with probability $1-\epsilon$, at each step. This corresponds to restricting the input space of the tree policy to only those that appear in the greedy strategy for a bounded number of steps: this helps the algorithm focus on most promising inputs rather than starting to explore randomly all input combinations. In practice, this helps to guide the search quickly towards the test objective, or to states that are nearby. The algorithm does eventually explore all inputs (after $M$ steps at a node) but at each newly created node, it again starts trying the greedy inputs. We also use reward shaping based on the distance remaining to the test objective inspired by [CIK$^+$19]: we give a state a high reward if its shortest path distance to the objective state in the requirement automaton is small.

We implemented the computation of greedy strategies, and its combination with MCTS. We present a small case study for which the combination of MCTS with greedy strategies allows one to reach the test objective, while plain MCTS or greedy strategies alone fail to find a solution.

**Related Works.** The notion of cooperative states have been used before in the setting of testing. These were used in [HJM18] in the context of offline test synthesis from timed automata, already inspired by a previous approach of test generation using games for transition systems in an untimed context [Ram98]. In [DLLN08a], 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 (a reachable test objective is not guar-

anteed to be found), 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 [VCG+08].

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 [JJ05] and TESTOR [MMS18] generate off-line test cases from formal specifications in various languages with IOLTS semantics, driven by test purposes described by automata. The tools JTorx [Bel10] and TorXakis [Tor] are improvement of TorX [TB03] and allow to generate and execute online test cases derived from various specification languages. In the context of timed models, Uppaal-TRON [HLM+08] is an online test generation tool for timed automata based on the real-time extension **rtioco** of the **ioco** conformance relation.

[MPRS11] 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. [AKKB18] uses Q-learning with the aim of covering as many states as possible; see also [LPZ+23]. In [RLPS20], 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. [THMT21] uses RL to learn short synchronizing sequences, where rewards correspond to the size of the powerset of states. In [PZAdS20], reinforcement learning was used to test shared memory programs. MCTS has been used for testing in various settings. In [ABCS20], it is used for testing video games using rewards to cover different areas in the game, but without automata specifications. Deep reinforcement learning has also been used for Android testing; see e.g. [RMCT21]. [FCP23] 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 [CIK+19], the approach is based on the distance to accepting states of Büchi automata; and in [VBB+21], 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.

# 2 Preliminaries

We first introduce traces that represent observable behaviours of reactive systems, then the automata models that recognize such traces and are used to formally specify requirements of such systems, together with related automata based notions.

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

We consider reactive systems that work as a succession of atomic steps: at each step, the environment first sets an input valuation $\nu^{\mathsf{in}}$, then the system immediately sets an output valuation $\nu^{\mathsf{out}}$. The valuation observed at this step is thus $\nu = \nu^{\mathsf{in}} \uplus \nu^{\mathsf{out}}$. A *trace* of the system is a sequence $\sigma = \nu_1 \cdot \nu_2 \cdots \nu_n$ of input and output valuations.

Internal variables may be used by the system to

compute outputs from the inputs and the internal state, but these are not observable to the outside.

**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 $\mathsf{AP}$ that need to be satisfied for the automaton to take that transition. For convenience, we handle input- and output valuations separately. Formally,

**Definition 2.0.1.** *An* automaton *is a tuple* $\mathcal{A} = \langle S = S^{in} \uplus S^{out}, s_{init}, AP, T, F\rangle$ *where* $S$ *is a finite set of states,* $s_{init} \in S^{in}$ *is the initial state,* $T \subseteq (S^{in} \times \mathcal{B}(AP^{in}) \times S^{out}) \uplus (S^{out} \times \mathcal{B}(AP^{out}) \times S^{in})$ *is a finite set of transitions, and* $F \subseteq S$ *is the set of accepting states.*

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

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

An automaton is *input-complete* if from any (reachable) state $s^{\mathsf{in}}$ and any valuation $\nu^{\mathsf{in}} \in 2^{\mathsf{AP}^{\mathsf{in}}}$, there is a transition $(s^{\mathsf{in}}, g^{\mathsf{in}}, s'^{\mathsf{out}})$ in $T$ such that $\nu^{\mathsf{in}} \models g^{\mathsf{in}}$. It is *output-complete* if a similar requirement holds for states in $S^{\mathsf{out}}$ and valuations in $\nu^{\mathsf{out}}$, 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 $s_1 = s_2$.

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

The set of *immediate predecessors* $\mathsf{Pre}_{\mathcal{A}}(B)$ of a set $B \subseteq S^{\mathsf{in}}$, and the set of its *immediate successors* $\mathsf{Post}_{\mathcal{A}}(B)$ are defined respectively as

$$\mathsf{Pre}_{\mathcal{A}}(B) = \bigcup_{s^{\mathsf{in}} \in B, \nu \in 2^{\mathsf{AP}}} \mathsf{Pre}_{\mathcal{A}}(s^{\mathsf{in}}, \nu),$$

$$\mathsf{Post}_{\mathcal{A}}(B) = \bigcup_{s^{\mathsf{in}} \in B, \nu \in 2^{\mathsf{AP}}} \mathsf{Post}_{\mathcal{A}}(s^{\mathsf{in}}, \nu).$$

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

$$\begin{aligned} &B \cup \mathsf{Pre}_{\mathcal{A}}(B) \cup \mathsf{Pre}_{\mathcal{A}}(B \cup \mathsf{Pre}_{\mathcal{A}}(B)) \\ &\quad \cup \mathsf{Pre}_{\mathcal{A}}\big(B \cup \mathsf{Pre}_{\mathcal{A}}(B) \cup \mathsf{Pre}_{\mathcal{A}}(B \cup \mathsf{Pre}_{\mathcal{A}}(B))\big) \\ &\quad \cup \cdots \end{aligned}$$

This correspond to an iterative computation of a sequence $(V_i)_{i \in \mathbb{N}}$, starting from $V_0 = \emptyset$ (which is why we get the *least* fixpoint) and such that $V_{i+1} = B \cup \mathsf{Pre}_{\mathcal{A}}(V_i)$. Observe, by induction on $i$, that from any state $s$ in $V_i$, there is a path to $B$ within $i$ steps. The sequence $(V_i)_{i \in \mathbb{N}}$ is non-decreasing, and its limit is the set of all states from which $B$ can be reached: from each state $s \in \mathsf{Pre}^*_{\mathcal{A}}(B)$, there is a finite trace $\sigma$ such that by reading $\sigma$ from $s$, one ends in $B$. Similarly, for each state $s' \in \mathsf{Post}^*_{\mathcal{A}}(B)$, there exists a state $s \in B$ and a trace $\sigma$ such that by reading $\sigma$ from $s$, one ends at $s'$.

Safety automata form a subclass of automata having a distinguished set $\mathbf{Error} \subseteq S^{\mathsf{in}}$ of error states that are non-accepting and absorbing (i.e., no transitions leave **Error**), and complement the set $F$ of accepting states in $S$ (i.e., $F = S \setminus \mathbf{Error}$). Those automata describe safety properties: nothing bad happened as long as **Error** is not reached.

The product of automata is defined as follows:

**Definition 2.0.2.** *Given automata* $\mathcal{A}_1 = \langle S_1^{in} \uplus S_1^{out}, s_{init,1}, AP_1, T_1, F_1 \rangle$ *and* $\mathcal{A}_2 = \langle S_2^{in} \uplus S_2^{out}, s_{init,2}, AP_2, T_2, F_2 \rangle$*, their* product $\mathcal{A}_1 \otimes \mathcal{A}_2$ *is an automaton* $\mathcal{A} = \langle S, s_{init}, AP, T, F \rangle$ *where* $S = (S_1^{in} \times S_2^{in}) \uplus (S_1^{out} \times S_2^{out})$, $s_{init} = (s_{init,1}, s_{init,2})$, $AP = AP_1 \cup AP_2$, $F = F_1 \times F_2$ *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. 2.0.1; and that completeness and determinism are preserved by the product. 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 **Error** in the product automaton is $(\mathbf{Error}_1 \times S_2^{\mathsf{in}}) \uplus (S_1^{\mathsf{in}} \times \mathbf{Error}_2)$, and thus inherits absorbance. The product of automata is commutative and associative, and can thus be generalized to an arbitrary number of automata.

We now consider an example of an automaton that will be used to illustrate other notions we define later in the paper.

**Example 2.0.1.** *Figure 1 displays an example of an automaton. For the sake of readability, 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, letting* $t$ *be an* **Error** *state 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* $o$.

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

**Example 2.0.2** (Factory Carriage Example)**.** *We consider a controller program in a factory automation system depicted in Fig. 2. 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.*

*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. 3. 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.*

# 3 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.

In the sequel, we use automata to describe requirements, and as models for implementations, with different assumptions. We identify a requirement with its complete deterministic safety automaton, and write $R$ for both.

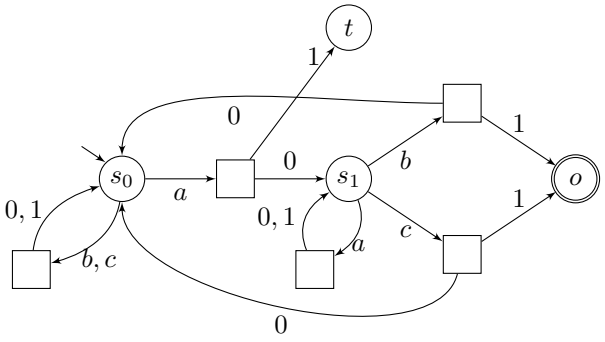


Figure 1: Example of a (deterministic) automaton expressing requirements.

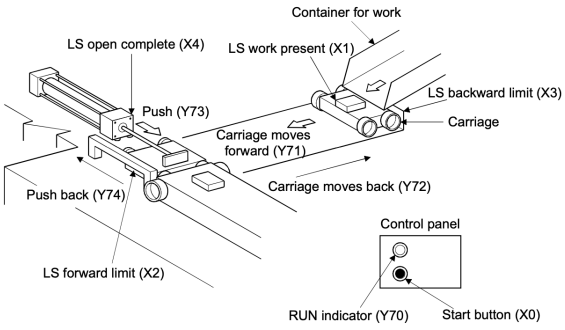


Figure 2: The carriage control system example from [Mit12].

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 $\otimes_{i \in I} R_i$.

**Definition 3.0.1.** *For any requirement $R$ defined by a complete deterministic automaton, and any finite trace $\sigma$, we write $\sigma$ **fails** $R$ if running $\sigma$ in $R$ from its initial state enters its error set* $\mathbf{Error}_R$.

For a trace $\sigma$ and a *set* of requirements $\mathcal{R}$ we write $\sigma$ **fails** $\mathcal{R}$ to mean that $\sigma$ **fails** $\mathcal{A}_\mathcal{R}$. Note the following simple facts, consequence of the definition of **Error** states in the product: if $\sigma$ **fails** $\mathcal{R}$ then $\sigma$ **fails** $R$ for at least one $R$ in $\mathcal{R}$; given $\mathcal{R}' \subseteq \mathcal{R}$, for any trace $\sigma$, if $\sigma$ **fails** $\mathcal{R}'$, then $\sigma$ **fails** $\mathcal{R}$.

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 $(2^{\mathsf{AP}})^*$. We assume that this system is a black box that proceeds as follows: at each step, an input valuation in $2^{\mathsf{AP}^{\mathsf{in}}}$ is provided to the system by the tester, and the system answers by producing an output valuation in $2^{\mathsf{AP}^{\mathsf{out}}}$ (on which the tester has no control). We make the following assumptions on the implementation: $I$ behaves as an unknown finite automaton over $\mathsf{AP}$; it is *input-complete*, meaning that any valuation in $2^{\mathsf{AP}^{\mathsf{in}}}$ can be set by the tester [1], and it is *output-deterministic*, meaning that any state in $S^{\mathsf{out}}$ has exactly one transition[2]. Last, we assume that from any input state of $I$, it is possible to reset $I$ to the unique initial state at any time. These properties ensure 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 $(2^{\mathsf{AP}})^*$.

The behaviour of $I$ is characterized by the set of all traces produced by the interaction between the tester and the system. Denote by $\mathrm{Tr}(I)$ the

[1] This is not restrictive since an implementation refusing some input valuations can be simulated by an input-complete implementation that would set a dedicated output variable to true in case of input refusal.

[2] Notice how *output-determinism* differs from standard *determinism*.

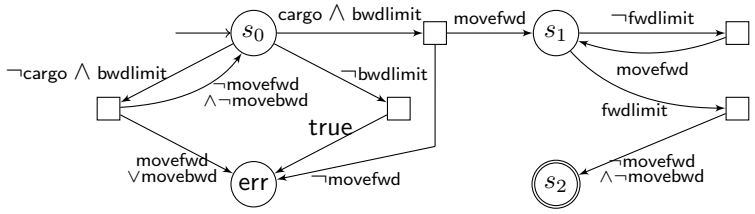


Figure 3: An automaton modeling requirements R1, R2, R3. For readability, we omitted some transitions in the figure: from all output states, there is an additional transition guarded by movebwd ∧ movefwd to err. Moreover, from each of the two rightmost output states, the negation of the guard of the only leaving transition goes to err as well. For instance, from $s_1$, reading ¬fwdlimit and then ¬movefwd, we end in err. Note that reading ¬bwdlimit in $s_0$ is also an error since this is not supposed to happen in this system.

set of traces that can be produced by $I$.

We now define what it means for an implementation to conform to a set of requirements:

**Definition 3.0.2.** *A system implementation $I \in \mathcal{I}$* conforms *to a set of requirements $\mathcal{R}$ on AP if for all $\sigma \in Tr(I)$, it is not the case that $\sigma$ **fails** $\mathcal{R}$.*

**Test Objectives.** In testing practice, each test case is related to a particular goal, e.g., derived from a coverage criterion. We formalise this now.

**Definition 3.0.3.** *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 $O \subseteq S^{in}_{\mathcal{A}_{\mathcal{R}}}$. A trace $\sigma$* covers *a test objective $O$, if the unique execution of $\mathcal{A}_{\mathcal{R}}$ on $\sigma$ ends in $O$.*

Notice that the more general case where a test objective is an automaton $A$ with a set of accepting locations Acc can be reduced to this one [3]. Indeed, it suffices to consider the product automaton of the test objective $A$ and the requirement automaton $\mathcal{A}_{\mathcal{R}}$, and consider as objective $O$ the set of states of the product in which $A$ is in Acc.

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

**Definition 3.0.4.** *The Test Problem*

**Input:** *a requirement set $\mathcal{R}$, a test objective $O$, and a 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 $O$, the algorithm 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 executing the covering traces of $I$, the tester will also check whether any generated trace fails $\mathcal{R}$. It will stop and report any such case.

Note that one can define some **Error** states as test objectives. In this case, the testing process will focus on generating traces that attempt to reach those errors, that is, on finding non-conformances.

In the next sections, we explain how to automatically synthesize test cases that build such traces while checking conformance of the implementation to the set of requirements.

**Example 3.0.1.** *In Example 2.0.2, we consider a test objective which is the singleton $\{s_2\}$ of Fig. 3.*

[3] This kind of automaton is sometimes called *test purpose* in the literature.

*In fact, reaching $s_2$ means that the implementation under test has made steps that are significant with respect to these requirements since this means that the carriage has successfully brought the cargo to the forward limit.*

# 4 Baseline Test Algorithms

Consider a set $\mathcal{R}$ of requirements, specified by a deterministic complete automaton $\mathcal{A}_\mathcal{R} = \langle S_\mathcal{R}, s^\mathcal{R}_{\mathsf{init}}, AP, T_\mathcal{R}, F_\mathcal{R}\rangle$, and an implementation whose behaviour could be modelled as an input-complete, output-deterministic finite automaton $I = \langle S_I, s^I_{\mathsf{init}}, AP, T_I, F_I\rangle$. 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 a black box to the tester.

Consider a particular test objective $O \subseteq S^{\mathsf{in}}_{\mathcal{A}_\mathcal{R}}$. Let $\mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O) = \mathsf{Pre}^*_{\mathcal{A}_\mathcal{R}}(O)$ 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 **Error** state (or both if $O$ contains **Error** 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 $\nu^{\mathsf{in}}$; this input valuation is fed to the implementation, which returns an output valuation $\nu^{\mathsf{out}}$ and moves to a new state $t^I$; the resulting valuation $\nu^{\mathsf{in}} \cup \nu^{\mathsf{out}}$ 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 $t^I$, unless we detect that a test objective or an **Error** state is reached.

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 part corresponding to $\mathcal{A}_\mathcal{R}$), 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.

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

Before explaining how we define test algorithms, we introduce some vocabulary to describe the configuration where the testing process ends. Assume that we have generated a trace $\sigma$ by interacting with $I$ from its initial state. Let $s_\sigma$ denote the state of $\mathcal{A}_\mathcal{R}$ reached after reading $\sigma$ from the initial state. Four cases may occur:

- if $s_\sigma \in O$, then $\sigma$ is a *covering trace* for $O$;
- if $s_\sigma \in \mathbf{Error}_{\mathcal{A}_\mathcal{R}}$, then $\sigma$ is an *error trace*;
- if $s_\sigma \notin \mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O)$, then $\sigma$ is *inconclusive*;
- otherwise, $s_\sigma \in \mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O) \setminus O$ and $\sigma$ 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 $O \cap \mathbf{Error}_{\mathcal{A}_\mathcal{R}} \neq \emptyset$. In the third case, no matter how we extend $\sigma$, we will never cover $O$; so the tester should stop and start again to look for another trace by resetting $I$. In the last case, $\sigma$ is active in the sense that it might still be possible to try to extend $\sigma$ to reach the objective.

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

**Example 4.0.1.** *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. 4. Their product is represented to the right of Fig. 4. There is a covering trace in this case since the input sequence ab generates the trace* $(a0b1)$ *in* $I_1$*, and this reaches the state* $o$ *in* $\mathcal{A}_\mathcal{R}$*.*

*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* **Error** *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* $o$*).*

We start by formalizing the naive uniform test approach, and then cast the problem as a reinforcement-learning problem.

## 4.1 Naive Uniform Testing

We present a simple test algorithm implemented in tools such as TorXakis [Tor]. Let $\mathsf{CoReach}^{\mathsf{in}}(\mathcal{A}_\mathcal{R}, O)$ denote the set of pairs $(s, \nu^{\mathsf{in}})$ where $s$ is a state of $\mathcal{A}_\mathcal{R}$, and $\nu^{\mathsf{in}}$ is an input valuation for which there exists some output valuation $\nu^{\mathsf{out}}$ such that $\mathsf{Post}_{\mathcal{A}_\mathcal{R}}(s, \nu) \in \mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O)$, where $\nu = \nu^{\mathsf{in}} \cup \nu^{\mathsf{out}}$. In fact, after observing trace $\sigma$ ending in state $s$ of $\mathcal{A}_\mathcal{R}$, the tester has no reason to give an input $\nu^{\mathsf{in}}$ such that $(s, \nu^{\mathsf{in}}) \notin \mathsf{CoReach}^{\mathsf{in}}(\mathcal{A}_\mathcal{R}, O)$: such an input would lead to a trace that is inconclusive, and the objective would not be reachable regardless of $I$.

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 $\{\nu^{\mathsf{in}} \mid (s, \nu^{\mathsf{in}}) \in \mathsf{CoReach}^{\mathsf{in}}(\mathcal{A}_\mathcal{R}, O)\}$. We observe the output $\nu^{\mathsf{out}}$ given by $I$, and update $s$ as $\mathsf{Post}_{\mathcal{A}_\mathcal{R}}(s, \nu^{\mathsf{in}} \cup \nu^{\mathsf{out}})$. 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 $s \in \mathbf{Error}_{\mathcal{A}_\mathcal{R}}$, 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.

Note that it is possible to generate inconclusive traces since $\mathcal{A}_\mathcal{R}$ is not assumed to be output-deterministic. It is then possible to have $(s, \nu^{\mathsf{in}}) \in \mathsf{CoReach}^{\mathsf{in}}(\mathcal{A}_\mathcal{R}, O)$ and for some $\nu^{\mathsf{out}}$ and $\nu'^{\mathsf{out}}$, $\mathsf{Post}_{\mathcal{A}_\mathcal{R}}(s, \nu^{\mathsf{in}} \cup \nu^{\mathsf{out}}) \in \mathsf{CoReach}^{\mathsf{in}}(\mathcal{A}_\mathcal{R}, O)$ and $\mathsf{Post}_{\mathcal{A}_\mathcal{R}}(s, \nu^{\mathsf{in}} \cup \nu'^{\mathsf{out}}) \notin \mathsf{CoReach}^{\mathsf{in}}(\mathcal{A}_\mathcal{R}, O)$ (see e.g., Fig. 1). Some conformant implementation $I$ can indeed return $\nu'^{\mathsf{out}}$, producing an inconclusive trace.

Let this test algorithm be called $\mathsf{uniform}$. For any bound $K$, let $\mathsf{uniform}_K$ be the uniform testing algorithm in which we stop each run after $K$ steps, so that the generated traces have length at most $K$. This algorithm is almost-surely complete:

**Lemma 4.0.1.** *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* $\mathsf{uniform}_K$ *finds a covering trace with probability* 1*.*

*Proof.* Assume there exists a covering trace $\sigma$ in $\mathcal{A}_\mathcal{R} \otimes I$, and let $K$ be the length of $\sigma$. When playing $\mathsf{uniform}_K$ *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 $2^{-|\mathsf{AP}^{\mathsf{in}}|}$. So at each restart, the probability of choosing exactly $\sigma$ is $2^{-|\mathsf{AP}^{\mathsf{in}}| \cdot |\sigma|}$. Therefore, $\sigma$ is picked eventually with probability 1. □

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

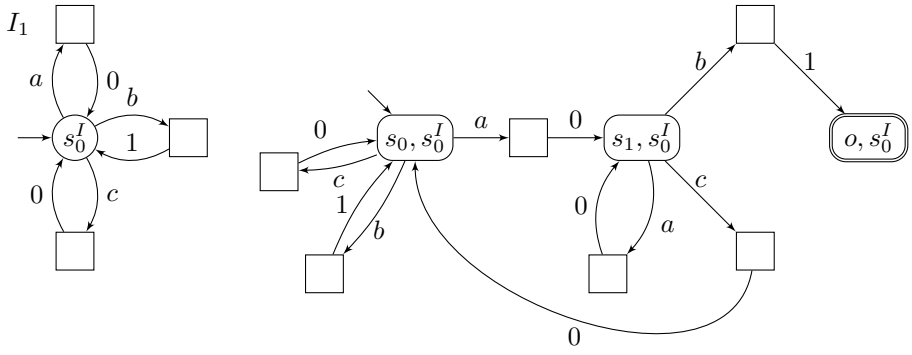


Figure 4: An implementation $I_1$ and its product with the automaton of Fig. 1

## 4.2 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. Note that we will consider *minimizing* the reward for reasons that will be clear later.

Notice that we do not assign a particular reward to error traces. In fact, we assume that the goal of the tester is to produce a covering trace, while monitoring all traces seen on the way for $\mathcal{R}$. If an error trace is seen, then we simply report it. Furthermore, it is possible to choose an objective in $\mathbf{Error}_{\mathcal{A}_\mathcal{R}}$ in which case the test strategy will try to reach an error state.

Reinforcement learning is a set of techniques that can be used to learn strategies that maximize the reward in games [SB18]. In this paper, we use Monte Carlo Tree Search [Cou06, KS06].

### 4.2.1 Monte-Carlo Tree Search.

Monte-Carlo Tree Search (MCTS) is a RL technique to search for good moves in games. It consists in exploring the 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) [KS06] which is standard in many applications. At a given node of the tree, if $n$ denotes the number of total visits to this node, and $n_i$ the number of times the $i$-th child node is visited (corresponding to the $i$-th move from the parent node), and $r_i$ the current average reward

of the $i$-th child, we define the score of the $i$-th child as $r_i + c\sqrt{\ln(n)/n_i}$ for some constant $c$. The UCT policy consists in choosing the child with the best score. Intuitively, this score is the average reward $r_i$ biased in order to make sure that each child node is visited frequently enough. In fact, the second term of the score is only relevant when $n_i$ is small. If the goal is to maximize the average score, then $c > 0$, and the UCT policy picks the child node with the maximal score; if, as in this work, we want to minimize the reward, then one chooses $c < 0$ and the policy picks the child node with minimal score.

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 reward values for 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. Last, we consider $\mathsf{uniform}_K$ (from Section 4.1) as the roll-out policy. Note that choosing the inputs uniformly in the simulation phase is standard. Here, we simply improve this by sampling over inputs that remain in the coreachable set.

#### 4.2.2 Reward Shaping: Accelerating Convergence.

One technique that is used to help reinforcement-learning algorithms converge faster is *reward shaping* [NHR99], 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 [CIK+19]. We now describe how we apply this to our setting.

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

$$C_i = \mathsf{Pre}_{\mathcal{A}_\mathcal{R}}(C_{i-1}) \setminus (C_0 \cup \ldots \cup C_{i-1}).$$

We have $\bigcup_{i \geq 0} C_i = \mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O)$. In fact, each $C_i$ is the set of states that are at distance $i$ from some state in $O$ (in the sense that $\mathcal{A}_\mathcal{R}$ contains a run of length $i$ to $O$).

We consider two ways of assigning rewards to simulation traces. Let $m$ be maximal such that $C_m \neq \emptyset$. Let us define $C_{m+1} = S_{\mathcal{A}_\mathcal{R}} \setminus \mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O)$, that is, all states that are not coreachable. We assign each trace $\sigma$ whose run in $\mathcal{A}_\mathcal{R}$ ends in $s^\mathcal{R}$ the reward $\mathsf{lastreward}(\sigma) = k$ 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 $r_0, r_1, r_2, \ldots, r_{K-1}$ denote the sequence of rewards encountered during simulation (these are the rewards of the prefixes of the trace $\sigma$). 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

$$\mathsf{disc\text{-}reward}_\gamma(\sigma) = r_{K-1} \cdot \sum_{i=0}^{K-1} \gamma^i \cdot r_i,$$

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 simulation has small rewards in the first steps.

#### 4.2.3 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 4.0.2.** *The basic MCTS algorithm is* complete*: 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 the basic MCTS with simulation bound $K$ finds a covering trace.*

Note that 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$_\gamma$. Moreover, as for uniform, it is possible not to fix $K$, but increase it slowly towards infinity.

Note that the basic MCTS algorithm is also a baseline since it can be obtained by combining known results from the literature; similar algorithms have been considered *e.g.* [VRC06, KS21].

# 5 Greedy Strategies and Improved Test Algorithms

In this section, we describe our original test algorithms. We consider a game-theoretic view of online testing, define particular strategies for the tester, and 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.

## 5.1 Controllable Predecessor and Successor Operators.

Suppose we are given requirements specified as the complete deterministic safety automaton $\mathcal{A}_\mathcal{R} = \langle S_{\mathcal{A}_\mathcal{R}}, s^{\mathcal{A}_\mathcal{R}}_{\mathsf{init}}, AP, T_{\mathcal{A}_\mathcal{R}}, F_{\mathcal{A}_\mathcal{R}} \rangle$ with $\mathbf{Error}_{\mathcal{A}_\mathcal{R}} = S_{\mathcal{A}_\mathcal{R}} \setminus F_{\mathcal{A}_\mathcal{R}}$, and a test objective $O \subseteq S^{\mathsf{in}}_{\mathcal{A}_\mathcal{R}}$. We consider the game $(\mathcal{A}_\mathcal{R}, O \cup \mathbf{Error}_{\mathcal{A}_\mathcal{R}})$ played between the *tester*, playing the role of the environment, and the *system*. The objective of the tester is to reach $O$ or to reveal a non-conformance (that is, reach $\mathbf{Error}_{\mathcal{A}_\mathcal{R}}$); it controls the input valuations in $2^{\mathsf{AP}^{\mathsf{in}}}$, and observes the output valuations in $2^{\mathsf{AP}^{\mathsf{out}}}$ 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 $B \subseteq S^{\mathsf{in}}_{\mathcal{A}_\mathcal{R}}$ is the set

$$\mathsf{CPre}_{\mathcal{A}_\mathcal{R}}(B) = \{ s \in S^{\mathsf{in}} \mid \exists \nu^{\mathsf{in}} \in 2^{\mathsf{AP}^{\mathsf{in}}}.\ \forall \nu^{\mathsf{out}} \in 2^{\mathsf{AP}^{\mathsf{out}}}.\ \mathsf{Post}_{\mathcal{A}_\mathcal{R}}(s, \nu^{\mathsf{in}} \cup \nu^{\mathsf{out}}) \in B \cup \mathbf{Error}_{\mathcal{A}_\mathcal{R}} \}.$$

Thus, from each state $s \in \mathsf{CPre}_{\mathcal{A}_\mathcal{R}}(B)$, the tester can select some valuation $\nu^{\mathsf{in}}$ such that whatever the output $\nu^{\mathsf{out}}$ 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 $\mathsf{CPre}^*_{\mathcal{A}_\mathcal{R}}(B) = \mathsf{lfp}(\lambda X.(B \cup$

$\mathsf{CPre}_{\mathcal{A}_\mathcal{R}}(X)$). 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.

**Example 5.0.1.** *In Figure 5, we have* $\mathsf{CPre}_{\mathcal{A}_\mathcal{R}}(O) = \{s_0, s_0'\}$ *and* $\mathsf{CPre}^*_{\mathcal{A}_\mathcal{R}}(O) = \{s_0, s_0'\} \cup O$.

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

A trace $\sigma = \nu_1 \cdot \nu_2 \cdots \nu_n$ is *compatible* with a strategy $\mathsf{f}$ from a state $s$ if there exists a sequence of states $q_0, q_1, q_2, \ldots, q_n$ such that $q_0 = s$ and for all $0 \leq i < n$, $q_i \xrightarrow{\nu_{i+1}} q_{i+1}$, and $\nu^{\mathsf{in}}_{i+1} \in \mathsf{f}(\nu_1 \cdots \nu_i)$. We write $\mathsf{Outcome}(\mathsf{f}, s)$ to denote the set of all traces compatible with $\mathsf{f}$ from $s$, also called the *outcomes* of $\mathsf{f}$ from $s$.

## 5.2 Winning and Cooperative Strategies

For a given set $B \subseteq S^{\mathsf{in}}$, a strategy $\mathsf{f}$ for the tester is *$B$-winning* from state $s$ if all its outcomes from $s$ eventually reach $B \cup \mathbf{Error}_{\mathcal{A}_\mathcal{R}}$. A state $s$ is $B$-winning if the tester has a winning strategy from $s$. The set of winning states can be computed using $\mathsf{CPre}^*$:

**Lemma 5.0.1.** *For all $B \subseteq S^{in}$, there exists a strategy $\mathsf{f}$ such that from all states $s \in \mathsf{CPre}^*_{\mathcal{A}_\mathcal{R}}(B)$, all $\mathsf{Outcome}(\mathsf{f}, s)$ eventually reach $B \cup \mathbf{Error}_{\mathcal{A}_\mathcal{R}}$.*

*Proof.* Consider the computation of

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

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

The strategy $\mathsf{f}$ provided by Lemma 5.0.1 is a *winning strategy* in the game $(\mathcal{A}_\mathcal{R}, B \cup \mathbf{Error}_{\mathcal{A}_\mathcal{R}})$.

One could look for a winning strategy in $(\mathcal{A}_\mathcal{R}, O \cup \mathbf{Error}_{\mathcal{A}_\mathcal{R}})$ in order to use it as a test algorithm. This approach was considered in some works (e.g., in [DLLN08b]). However, in most cases, $\mathsf{CPre}^*_{\mathcal{A}_\mathcal{R}}(O)$ 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.

**Cooperative Steps.** For given set $B \subseteq S^{\mathsf{in}}$, consider $\mathsf{Pre}_{\mathcal{A}_\mathcal{R}}(B)$, the set of immediate predecessors of $B$: by definition from these states, there exists a pair of valuations $(\nu^{\mathsf{in}}, \nu^{\mathsf{out}})$ 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 $\nu^{\mathsf{in}}$ for which *there exists* $\nu^{\mathsf{out}}$ which moves the system into $B$. In fact, when attempting to reach $B$, if the current state is not in $\mathsf{CPre}^*_{\mathcal{A}_\mathcal{R}}(B)$, then choosing such a $\nu^{\mathsf{in}}$ is a good choice.

Let us call a strategy $\mathsf{f}$ *$B$-cooperative* if for all $s \in \mathsf{Pre}_{\mathcal{A}_\mathcal{R}}(B)$, and all $\nu^{\mathsf{in}} \in \mathsf{f}(s)$, there exists $\nu^{\mathsf{out}}$ such that $\mathsf{Post}_{\mathcal{A}_\mathcal{R}}(s, \nu^{\mathsf{in}} \cup \nu^{\mathsf{out}}) \in B$.

**Example 5.0.2.** *In Fig. 5, in $s_0^c$ the $O$-cooperative strategy $\mathsf{f}$ is represented by a bold arrow. Choosing this input valuation may lead to $O$ at the next step if the system "cooperates" by choosing the right output valuation, while choosing the input valuation outside $\mathsf{f}(s)$ 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 $\mathcal{A}_\mathcal{R}$ only against

some system strategies. We will then show how to obtain $\epsilon$-greedy strategies that can be applied against any system strategy, and use these as heuristics to improve over the basic MCTS test algorithm.

## 5.3 Greedy Strategy

Consider $W_0 = \mathsf{CPre}^*_{\mathcal{A}_\mathcal{R}}(O)$, which is non-empty since it contains $O$. Let $\mathsf{f}_0$ be a $O$-winning strategy given by Lemma 5.0.1. If $s_{\mathsf{init}}$ belongs to $W_0$, then $\mathsf{f}_0$ ensures reaching $O \cup \mathbf{Error}_{\mathcal{A}_\mathcal{R}}$ from $s_{\mathsf{init}}$ against all system strategies. In this case, we stop and return $\mathsf{f}_0$ as the greedy strategy.

The interesting and more frequent case is when $s_{\mathsf{init}} \notin W_0$. In this case, we inductively define an increasing sequence of sets of states $W_0, W_1, \ldots, W_n$ 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 $\mathsf{Coop}_{i+1} = \mathsf{Pre}_{\mathcal{A}_\mathcal{R}}(W_i) \setminus W_i$ be the set of immediate predecessors of $W_i$ deprived of states that have been already seen. Define $\mathsf{f}^c_{i+1}$ as a $W_i$-cooperative strategy. Then, let $W_{i+1} = \mathsf{CPre}^*_{\mathcal{A}_\mathcal{R}}(\mathsf{Coop}_{i+1} \cup W_i)$, and consider a corresponding winning strategy $\mathsf{f}^w_{i+1}$. Let us denote by $\mathsf{f}_{i+1}$ the pointwise union of the strategies $\mathsf{f}^w_{i+1}$ and $\mathsf{f}^c_{i+1}$: for all $s \in S^{\mathsf{in}}$, if $s \in \mathsf{Coop}_{i+1}$, then $\mathsf{f}_{i+1}(s) = \mathsf{f}^c_{i+1}(s)$; if $s \in W_{i+1} \setminus \mathsf{Coop}_{i+1}$, then $\mathsf{f}_{i+1}(s) = \mathsf{f}^w_{i+1}(s)$; and $\mathsf{f}_{i+1}$ is defined arbitrarily otherwise.

We stop this sequence whenever $\mathsf{Coop}_i$ becomes empty. Notice that we have

$$\mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O) = \mathsf{Pre}^*_{\mathcal{A}_\mathcal{R}}(O) = \bigcup_i W_i,$$

and that the sequence $(W_i)_i$ is increasing.

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 $\mathsf{Coop}_i$ to get closer to $O$ in this hierarchy. Because all states of $\mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O)$ belong to some $W_i$, in particular, if $O$ is reachable from $s_{\mathsf{init}}$ (i.e., $s_{\mathsf{init}} \in \mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O)$), there exists some index $k$ such that $s_{\mathsf{init}}$ in $W_k$. For a state $s$, we call the *rank* of $s$ and note $\mathsf{rank}(s)$ the smallest index $i$ such that $s \in W_i$.

We denote by $\mathsf{f}_{\mathsf{greedy}}$ the strategy defined by $\mathsf{f}_{\mathsf{greedy}}(s) = \mathsf{f}_{\mathsf{rank}(s)}(s)$ for all $s \in \mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O)$; and arbitrarily for other states. We call this the *greedy strategy*.

Notice that none of the winning strategies ($\mathsf{f}^w_i$ or $\mathsf{f}_0$) composing $\mathsf{f}$ contains loops. However, since $\mathsf{f}$ is also composed of cooperative strategies, loops may occur due to outputs reaching states with higher ranks in the $(W_i)_i$ hierarchy. In other terms, $\mathsf{f}_{\mathsf{greedy}}$ does not guarantee reaching $O \cup \mathbf{Error}_{\mathcal{A}_\mathcal{R}}$; and it may induce infinite loops against some system strategies.

The construction is illustrated in Fig. 5. States in $S^{\mathsf{in}}$ where the tester plays are represented by circles, while transient states in $S^{\mathsf{out}}$ where the system plays are represented by squares. In a state $s_i$ in $W_i$, the strategy $\mathsf{f}_i$ is illustrated by bold blue arrows. Black arrows represent those outputs that either reach a state in a set $W_k$ with higher rank or reach $S \setminus \mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O)$. For example, in $s_0 \in W_0$, the input in bold is winning since all subsequent outputs reach $O$. But the other input is not winning since one possible output loops back in $s^c_0$ thus in $W_1$, and the other one goes back to some higher rank $i$. A different situation is illustrated by $s^c_1$ in $\mathsf{Coop}_2$: after some input in the cooperative strategy, one output may reach $W_1$, but the other one reaches a state in $S \setminus \mathsf{CoReach}(\mathcal{A}_\mathcal{R}, O)$.

Intuitively, the strategy $\mathsf{f}_{\mathsf{greedy}}$ requires minimal cooperation from the system to reach $O$ or $\mathbf{Error}_{\mathcal{A}_\mathcal{R}}$ from $s_{\mathsf{init}}$.

**Example 5.0.3.** *Consider the automaton of Fig. 1 again. Here we have $W_0 = \{o\}$, $\mathsf{Coop}_0 = \{s_0, s_1\}$ while $\mathsf{f}_{\mathsf{greedy}}(s_0) = \{a\}$ and $\mathsf{f}_{\mathsf{greedy}}(s_1) = \{b, c\}$. The implementation given in Fig. 6 admits a covering trace which starts with $(b, 0)$. However, because $b \notin \mathsf{f}_{\mathsf{greedy}}(s_0)$, this trace cannot be found by the greedy strategy.*

As the previous example shows, the greedy strategy does not always guarantee progress to-

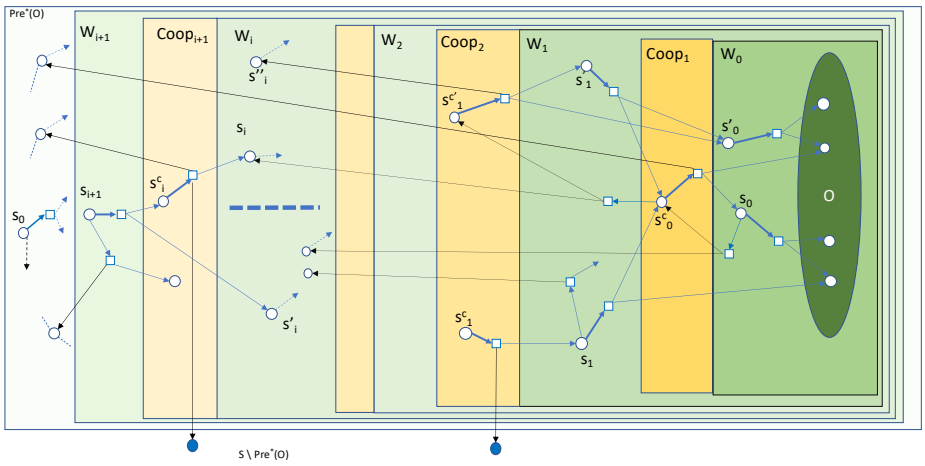


Figure 5: Construction of the $(W_i)_i$ hierarchy and cooperative strategies

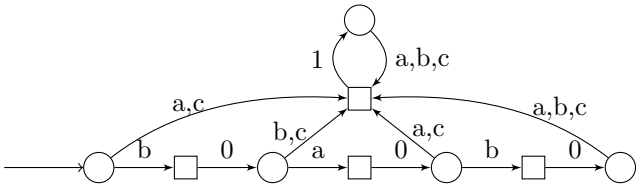


Figure 6: An implementation which contains the trace $(b,0),(a,0),(b,0)$ covering $\{o\}$ in the automaton of Fig. 1. This is the only covering trace.

wards the objective at each step; we rather see it as a best effort heuristic which might or might not help reaching the objective. We use the greedy strategy in different ways to improve online testing as explained next.

## 5.4 Purely greedy and $\epsilon$-greedy test algorithms

The greedy strategy can be used to define a randomized test algorithm, called the *pure greedy* algorithm, and denoted $\mathsf{greedy}$, as follows. Recall that after an active trace $\sigma$, the automaton $\mathcal{A}_{\mathcal{R}}$ is in a state $s_\sigma \in \mathsf{CoReach}(\mathcal{A}_{\mathcal{R}}, O)$. In the uniform strategy an input $\nu^{\mathsf{in}}$ is chosen uniformly in $\{\nu^{\mathsf{in}} \mid (s_\sigma, \nu^{\mathsf{in}}) \in \mathsf{CoReach}^{\mathsf{in}}(\mathcal{A}_{\mathcal{R}}, O)\}$. A simple modification consists in replacing this choice with a uniform choice in $\mathsf{f}(s_\sigma)$ to restrict the domain to the transitions selected by the greedy strategy. If $s_\sigma$ is in $W_i$ for some $i$, following this strategy will inevitably lead either to $\mathsf{Coop}_i$ if $i > 0$, and to $O$ if $i = 0$, or possibly to **Error** if $I$ is non-conformant. Once in $\mathsf{Coop}_i$, the tester uses $\mathsf{f}_{\mathsf{greedy}}$ because there is a possibility of entering $W_{i-1}$; but the implementation, even if conformant, is not forced to be cooperative, and may go back to some $W_j$ with $j \geq i$. For $K > 0$, let $\mathsf{greedy}_K$ denote the test algorithm obtained from $\mathsf{greedy}$ by stopping each run after $K$ steps, and restarting another run.

The algorithm $\mathsf{greedy}_K$ may not be almost-surely complete for any $K > 0$ as we already saw. However, we can obtain a almost-surely complete algorithm $\epsilon$-$\mathsf{greedy}$ simply by randomizing between $\mathsf{uniform}$ and $\mathsf{f}_{\mathsf{greedy}}$: given $0 < \epsilon < 1$, this algorithm uses, at each state $s_\sigma$, $\mathsf{f}_{\mathsf{greedy}}(s_\sigma)$ with

probability $1-\epsilon$; and $\mathsf{uniform}$ with probability $\epsilon$.

In fact, if $O$ is reachable in $\mathcal{A}_{\mathcal{R}} \otimes I$, say, within $k < K$ steps, then, at each run, there is a positive probability that the $\epsilon\text{-}\mathsf{greedy}_K$ executes $\mathsf{uniform}$ for $k$ steps, while picking the right inputs, also with positive probability. Repeating this experiment makes sure that the right sequence will be chosen eventually with probability 1.

### 5.5 The Greedy-MCTS Test Algorithm

Here, we explain our main contribution which is an improvement of the Basic MCTS test algorithm of Section 4.2 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 $\epsilon\text{-}\mathsf{greedy}_K$ as the roll-out policy instead of $\mathsf{uniform}_K$, for some given $K$.

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 of the greedy strategy $\mathsf{f}_{\mathsf{greedy}}(s)$ at the first $M$ visits to that tree node; after the first $M$ visits to a given node, we fallback to the regular UCT policy, which covers the whole set of input valuations $\{\nu^{\mathsf{in}} \mid (s_\sigma, \nu^{\mathsf{in}}) \in \mathsf{CoReach}^{\mathsf{in}}(\mathcal{A}_{\mathcal{R}}, O)\}$. It should be noted that because the bound $M$ applies separately to each node, at any moment, there are always nodes (close to leaves) 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 the convergence of the test algorithm by introducing bias in probabilistic choices. In fact, similar biased UCT scores have been used *e.g.* in applications for the board game Go [GS11].

### 5.6 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. 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.

## 6 Case Study

### 6.1 Description of the System

We consider a system controlling a robot moving in a discrete 2D grid environment made of N=10 rooms placed horizontally, a *passageway*, each room being separated from the previous one by a door. Each room except the first one has a door to its left, and each room except the last one has a door to its right. The robot occupies a single discrete cell at any moment, and the **input** given to the system makes the robot move to one of the four diagonal neighboring cells. More precisely, the Boolean inputs are $\mathsf{AP}^{\mathsf{in}} = \{\mathsf{right}, \mathsf{up}\}$: the robot moves right one step if $\mathsf{right}$ holds; it moves left otherwise; it moves upwards if $\mathsf{up}$ holds; it moves downwards otherwise. The doors at all

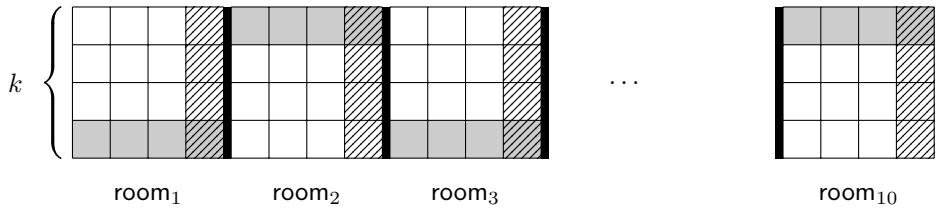


Figure 7: The passageway implementation of the rooms with $k = 4$. The shaded area is the open area, and the hatched area is the doorstep area. The thick black lines show the doors.

rooms are initially closed. In order to move to the next room, the robot must first visit an area called *open*, which opens the door, and another area called *doorstep* which is made of cells neighboring the door and the next room. Leaving the open area makes the door close again. Furthermore, moving towards a wall or a closed door causes a *collision*.

An example of such an implementation is shown in Fig. 7 where each room is modelled as a 4x4 grid. The open area here is alternatively the lowest or the highest row, and the doorstep is the rightmost column in each of the first nine rooms. The door at the right of a room is open if, and only if, the robot is inside the shaded area; and the robot can actually move to the next room (it is at the doorstep) if, and only if, it is also on the hatched area: so it can do so only at the bottom right cell of the first room; and top right cell of the second one, etc.

The **outputs** of the system are the following: the identifier of the room the robot is currently occupying ($\{\mathsf{room}_1, \ldots, \mathsf{room}_{10}\}$), whether the robot is in the open and doorstep areas, and whether it is currently in a collision. Thus, the implementation does not output the precise position of the robot; but only an indication about its position.

What is described in Figure 7 is one possible implementation. Our objective is to write abstract requirements that can be used for testing various implementations. A different implementation can use rooms of different sizes and shapes, have additional walls, or (deterministically) moving obstacles inside rooms, and consider different dynamics for the moves of the robot.

We considered a requirement automaton that describes properties of this system, imposing constraints both on the environment and on the program that controls the robot. We enforce that it is only possible to move to the next room if the door is open and the robot is at the doorstep; and that moving right when at doorstep and open, the next room is entered; while this is not the case when moving left from such a state. We also impose that in rooms $i$ with odd $i$, open cannot be reached by going up; this means that it must be on the bottom-most part of the room. In rooms $i$ with even $i$, the situation is reversed: one cannot reach open by going down. Fig. 8 shows a part of the automaton representation of this requirement corresponding to the robot being in room $i$. We are at state $m_0$ when the robot is not in the open area; at $m_1$ when it is in the open area but not at doorstep; and at $m_2$ when it is both in the open and doorstep areas. Intuitively, reaching the next room requires going from $m_0$ to $m_2$ (either directly, or via $m_1$).

Let us illustrate the inputs chosen by the greedy strategy here. First, consider the state $m_1$: choosing to go left can either go to error, to $m_0$, or to $m_1$; but none of the system outputs can make state move closer to the next room. On the other hand, going right, possible outcomes are $m_0$, $m_1$, or $m_2$: therefore, this is a cooperative state (belongs to some $\mathsf{Coop}_i$ in the computation of Section 5), and going right belongs to the greedy strategy. A similar situation arises at state $m_2$: going right leads possibly to the next room (although not surely due to collision outputs not

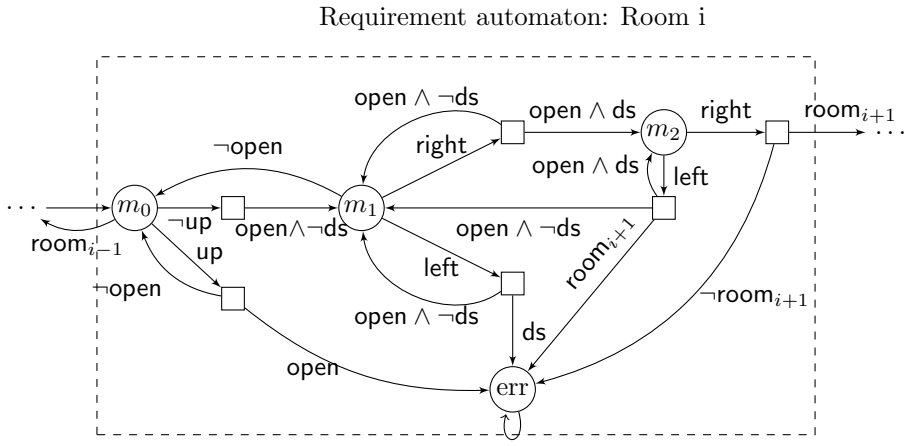


Figure 8: Part of the requirement automaton for the passageway example that corresponds to room i with odd $i \in (1, 10)$. There is a similar structure to the left, and to the right. Here, left is a shorthand for ¬right; ds is short for doorstep. Some transitions lead directly to input states, without intermediate output states (such as the transition from $m_1$ to $m_0$): this means that the transition is independent from the valuations of variables not mentioned on the guard. Some guards and transitions are omitted for clarity: all transitions that do not mention any variable of the form $\mathsf{room}_j$ are assumed to be guarded by $\mathsf{room}_i$. In fact, this part of the automaton corresponds to the robot being in room i. From all states fresh transitions guarded by $\mathsf{room}_{i-1}$ go to the corresponding state to the left of the present figure. Whenever any other $\mathsf{room}_j$ with $j \neq i-1, i$ is set to 1, this leads to **Error**. Furthermore, from any state, if collision holds, then we move to an absorbing state called collision (not shown here).

shown on the figure), but going left definitely cannot make the state move to the next room. Thus, the greedy strategy chooses to go right but not left.

To evaluate our test algorithms, we considered the implementation described above with the following non-conformance: in room 9, when the door is open, and at doorstep, moving right does not move the robot to the next room. We considered the test objective of reaching any cell in the last room without ever being in a collision.

Note that although our implementation only has 160 states (16 possible positions in 10 rooms), finding an execution reaching the last room is particularly difficult since a collision occurs whenever the robot moves towards a wall, and this happens very quickly in a uniform random test.

## 6.2 Test Algorithm Implementation

We implemented the test generation algorithm in Python where requirements are specified as deterministic finite automata specified as Verilog modules. These requirement modules are automatically translated to the AIGER format using Yosys and ABC), which can be read by the Abssynthe game solver. We implemented the greedy strategy computation in Abssynthe, but also the computation of the predecessor sequence $C_0, C_1, \ldots$ from Section 4.2.2 to compute rewards. The testing tool moreover uses the CUDD BDD library, and pyAIGER to read and execute the greedy strategy computed by Abssynthe, and to compute rewards. The tester communicates with the program under test via standard input and

output.

### 6.3 Experimental Results

We compared several algorithms on the described case study. These include the baseline algorithms $\mathsf{uniform}_K$, $\epsilon\text{-}\mathsf{greedy}_K$ (with $\epsilon = 0.25$), and the basic MCTS. We allowed each test algorithm 50 attempts to reveal the bug. At each attempt, we made 10,000 runs; each run starting from the initial state and making $K = 250$ steps. For uniform and pure greedy algorithms, this meant that we made a total number of 500,000 runs, each making 250 steps. For the basic MCTS, this meant that we started from scratch 50 times, and ran 10,000 runs, each with a roll-out of length 250.

The results of various algorithms on the described case study are given in Table 1. All three baseline algorithms failed at revealing the bug, and increasing the number of steps to $K = 1000$ did not change the outcome.

The Greedy-MCTS tester was more successful. We considered two variants. In the first one, we used the standard UCT tree policy and the greedy policy for the roll-outs (greedy rollout). Among the 50 attempts, this approach found a covering trace in 62.7% of the cases. The bug was revealed after 4662 runs in average among successful attempts (each attempt was stopped whenever a covering trace is found or when 10,000 runs are made). The second variant uses, moreover, the greedy policy inside the tree for the first M=30 visits at each node, and the UCT afterwards (greedy tree & rollout). The success rate was 100%, with only 1031 runs in average.

## 7 Conclusion

We presented an algorithm for online testing of reactive programs with respect to automata-based requirements based on an improvement of the Monte-Carlo Tree Search algorithm with heuristics. These heuristics are computed by a game-theoretic view of testing. While the game point of view has been explored before, we use it to improve the reinforcement learning approach. Our preliminary experimental results show that these heuristics can improve the testing time by guiding the search quickly to relevant states. This is especially the case when test objectives require long sequences that have low probability to be found by uniform random testing.

As future work, we plan to make a systematic study of this approach to evaluate its limits to usability in an industrial context. Targeting particular applications such as GUI testing (as [KS21]) is a possibility since the particular forms of temporal logic requirements might allow one to derive better heuristics tailored for the application at hand.

| **Algorithm** | **Success Rate** | **Average runs** |
|---|---|---|
| $\mathsf{uniform}_K$ | 0% | - |
| $\epsilon$-$\mathsf{greedy}_K$ | 0% | - |
| Basic MCTS | 0% | - |
| MCTS + greedy roll-out | 62.7% | 4662 |
| MCTS + greedy tree & roll-out | 100% | 1031 |
| TorXakis | 0% | - |

Table 1: Results of different algorithms on our case study. The success rate is the number of attempts that revealed the bug; while the average runs is the average number of runs made by each successful attempt before revealing the bug.

# References


[ABCS20] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. Enhancing the Monte Carlo tree search algorithm for video game testing. In *2020 IEEE Conference on Games (CoG)*, pages 25–32. IEEE, 2020.

[AKKB18] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. Reinforcement learning for android GUI testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating Test Case Design, Selection, and Evaluation*, pages 2–8, 2018.

[AO17] Paul Ammann and Jeff Offutt. *Introduction to Software Testing Edition 2*. Cambridge University Press, New York, NY, 2017.

[BB96] Mark R Blackburn and Robert D Busser. T-VEC: A tool for developing critical systems. In *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96*, pages 237–249. IEEE, 1996.

[Bei95] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[Bel10] Axel 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*, pages 266–270. Springer, 2010.

[Bol15] William Bolton. *Programmable logic controllers*. Newnes, 2015.

[BPW+12] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[CIK+19] Alberto Camacho, Rodrigo Toro Icarte, Toryn Q Klassen, Richard Anthony Valenzano, and Sheila A McIlraith. LTL and beyond: Formal languages for reward function specification in reinforcement learning. In *IJCAI*, volume 19, pages 6065–6073, 2019.

[Cou06] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[DLLN08a] Alexandre David, Kim Larsen, Shuhao Li, and Brian Nielsen. Cooperative testing of timed systems. In *4th Workshop on Model Based Testing (MBT'08)*, volume 220, pages 79–92, 12 2008.

[DLLN08b] Alexandre David, Kim Guldstrand Larsen, Shuhao Li, and Brian Nielsen. A game-theoretic approach to real-time system testing. In *Proceedings of the 2000 Design, Automation and Test in Europe (DATE'00)*, pages 486–491. IEEE Comp. Soc. Press, March 2008.

[FCP23] Roi Fogler, Itay Cohen, and Doron Peled. Accelerating black box testing with light-weight learning. In Georgiana Caltais and Christian Schilling, editors, *Model Checking Software*, pages 103–120, Cham, 2023. Springer Nature Switzerland.

[GS11] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[HJM18] Léo Henry, Thierry Jéron, and Nicolas Markey. Control strategies for off-line testing of timed systems. In María-del-Mar Gallardo and Pedro Merino, editors, *Model Checking Software - 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings*, volume 10869 of *Lecture Notes in Computer Science*, pages 171–189. Springer, 2018.

[HLM+08] Anders Hessel, Kim G Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, pages 77–117, 2008.

[JG16] Bertrand Jeannet and Fabien 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.

[JJ05] Claude Jard and Thierry Jéron. 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*, 7:297–315, 2005.

[KS06] Levente Kocsis and Csaba 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*, pages 282–293. Springer, 2006.

[KS21] Yavuz Köroglu and Alper Sen. Functional test generation from UI test scenarios using reinforcement learning for android applications. *Softw. Test. Verification Reliab.*, 31(3), 2021.

[LLGS18] Wenbin Li, Franck Le Gall, and Naum 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*, pages 77–89. Springer, 2018.

[LPZ+23] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping 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*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[MBTS04] Glenford J. Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[Mit12] Mitsubishi Electric Corporation. Mitsubishi programmable controller – Training manual, 2012. `https://dl.mitsubishielectric.com/dl/fa/document/manual/school_text/sh081123eng/sh081123enga.pdf`.

[MMS18] Lina Marsso, Radu Mateescu, and Wendelin Serwe. Testor: A modular tool for on-the-fly conformance test case generation. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 211–228, Cham, 2018. Springer International Publishing.

[MPRS11] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: A tool for automatic black-box testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 1013–1015, New York, NY, USA, 2011. Association for Computing Machinery.

[NHR99] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287. Citeseer, 1999.

[PZAdS20] Nícolas Pfeifer, Bruno V. Zimpel, Gabriel A. G. Andrade, and Luiz 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)*, pages 538–543, 2020.

[Ram98] Solofo Ramangalahy. Strategies for conformance testing. Technical Report MPI-I-98-010, Max Planck Institut Für Informatik, May 1998.

[RLPS20] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1410–1421, 2020.

[RMCT21] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. Deep reinforcement learning for black-box testing of android apps. *ACM Trans. Softw. Eng. Methodol.*, 2021.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[TB03] GJ Tretmans and Hendrik Brinksma. Torx: Automated model-based testing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, 2003.

[THMT21] Uraz Cengiz Türker, Robert M. Hierons, Mohammad Reza Mousavi, and Ivan Y. Tyukin. Efficient state synchronisation in model-based testing through reinforcement learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 368–380, 2021.

[Tor] Torxakis. `https://github.com/torxakis`.

[Tre96] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Softw. Concepts Tools*, 17(3):103–120, 1996.

[TvdL19] Jan Tretmans and Piërre van de Laar. Model-based testing with TorXakis. In *Central European Conference on Information and Intelligent Systems*, pages 247–258. Faculty of Organization and Informatics Varazdin, 2019.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, 22(5):297–312, 2012.

[VBB+21] Alvaro Velasquez, Brett Bissey, Lior Barak, Andre Beckus, Ismail Alkhouri, Daniel Melcer, and George Atia. Dynamic automaton-guided reward shaping for Monte Carlo tree search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(13):12015–12023, 2021.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, pages 39–76, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[VRC06] Margus Veanes, Pritam Roy, and Colin Campbell. Online testing with reinforcement learning. In Klaus Havelund, Manuel Núñez, Grigore Roşu, and Burkhart Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, pages 240–253, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Yan04] Mihalis Yannakakis. Testing, optimization, and games. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 78–88, 2004.