Online Testing with Reinforcement Learning

Margus Veanes[1], Pritam Roy[2] *, and Colin Campbell[1]

[1] Microsoft Research, Redmond, WA, USA
{margus,colin}@microsoft.com
[2] University of California, Santa Cruz, USA
pritam@soe.ucsc.edu

Abstract. Online testing is a practical technique where test derivation and test execution are combined into a single algorithm. In this paper we describe a new online testing algorithm that optimizes the choice of test actions using Reinforcement Learning (RL) techniques. This provides an advantage in covering system behaviors in less time than with a purely random choice of test actions. Online testing with conformance checking is modeled as a $1\frac{1}{2}$-player game, or Markov Decision Process (MDP), between the tester as one player and the implementation under test (IUT) as the opponent. Our approach has been implemented in C#, and benchmark results are presented in the paper. The specifications that generate the tests are written as model programs in any .NET language such as C# or VB.

1 Introduction

Many software systems are reactive. The behavior of a reactive system, especially when distributed or multithreaded, can be nondeterministic. For example, systems may produce spontaneous outputs like asynchronous events. Factors such as thread scheduling are not entirely under the control of the tester but may still affect the behavior observed. In these cases, a test suite generated offline may be infeasible, since all of the observable behaviors would have to be encoded a priori as a decision tree, and the size of such a decision tree can be very large.

Online testing (also called on-the-fly testing) can be more appropriate than offline tests for reactive systems. The reason is that with online testing the tests may be dynamically adapted at runtime, effectively pruning the search space to include only those behaviors actually observed instead of all possible behaviors. The interaction between tester and implementation under test (IUT) is seen as a game [1] where the tester chooses moves based on the observed behavior of the implementation under test. Only the tester is assumed to have a goal; the other player (the IUT) is unaware that it is playing. This kind of game is known in the literature as a $1\frac{1}{2}$-player game [6].

Online testing is a form of *model-based testing (MBT)*, where the tester uses a specification (or *model*) of the system's behavior to guide the testing and to detect the discrepancies between the IUT and the model. It is an established technique, supported in tools like TorX [18] and Spec Explorer [20]. For the purposes of this paper, we express

* Part of this work was done during the author's summer internship at Microsoft Research.

the model as a set of guarded update rules that operate on an abstract state. This formulation is called a *model program*. Both the IUT and the model are viewed as *interface automata* [8] in order to establish a a formal conformance relation between them.

We distinguish between moves of the tester and moves of the IUT. The actions available to the tester are called *controllable* actions. The IUT's responses are *observable* actions. A *conformance failure* occurs when the IUT rejects a controllable action produced by the model or when the model rejects an observable action produced by the IUT.

A principal concern of online testing is the *strategy* used to choose test actions. A poor strategy may fail to provoke behaviors of interest or may take an infeasible amount to time to achieve good coverage. One can think of strategy in economic terms. The cost of testing increases with the number of test runs and the number of steps per run. We want to minimize the number of steps taken to achieve a given level of coverage for the possible behaviors. Exhaustive coverage is often infeasible. Instead, we strive for the best coverage possible within fixed resource constraints. The main challenge is to choose actions that minimize backtracking, since resetting the IUT to its initial state can be an expensive operation.

A purely random strategy for selecting test actions can be wasteful in this regard, since the tester may repeat actions that have already been tested or fail to systematically explore the reachable model states. A random strategy cannot benefit from remembering actions chosen in previous runs.

In this paper we propose an algorithm for online testing, using the ideas from *Reinforcement Learning (RL)* [16, 12]. RL techniques address some of the drawbacks of random action selection. Our algorithm is related to the anti-ant algorithm introduced in [13], which avoids the generation of redundant test cases from UML diagrams.

RL refers to a collection of techniques in which an *agent* makes moves (called *actions*) with respect to the *state* of an environment. Actions are associated with *rewards* or *costs* in each state. The agent's goal is to choose a sequence of actions to maximize expected reward or, equivalently, to minimize expected cost.

The history needed to compute the strategy is encoded in a data structure called a "Test-Trace Graph (TTG)". We compare several such strategies below. The results show that a greedy strategy (*LeastCost*) has a suboptimal solution. The probability of reaching a failure state does not change with a purely randomized strategy (*Random*), though the probability reduces monotonically in a randomized greedy strategy (*RandomizedLeastCost*). This is because the probability in the latter case is negatively reinforced by the number of times a failure state has been visited, whereas it remains same in the former case.

The contributions of this paper are the following:

- We transform the online testing problem into a special case of reinforcement learning where the frequencies of various abstract behaviors are recorded. This allows us to better choose controllable actions.
- We show with benchmarks that an RL-based approach can significantly outperform random action selection.

The rest of the paper is organized as follows. In Section 2 we provide definitions for model programs, interface automata and a conformance relation. In Section 3 we give a detailed description of the algorithm. In Section 4 we give the experimental results from our benchmarks. We discuss related work in Section 5 and open problems and future work in Section 6.

2 Testing Theory

In model-based testing a tester uses a specification for two purposes. One is *conformance checking*: to decide if the IUT behaves as expected or specified. The other is *scenario control*: which actions should be taken in which order and pattern. Model-based testing is currently a growing practice in industry. In many respects the second purpose is the main use of models to drive tests and relates closely to test scenarios is traditional testing. However, with a growing complexity and need for protocol level testing and interaction testing, the first purpose is gaining importance.

Formally, model programs are mapped (unwound) to interface automata in order to do conformance checking. The conformance relation that is used can be defined as a form of alternating refinement. This form of testing is provided by the Spec Explorer tool, see e.g. [20].

2.1 Model programs as Specifications

States are *memories* that are finite mappings from (memory) locations to a fixed universe of values. By an update rule we mean here a finite representation of a function that given a memory (state) produces an updated memory (state). An update rule p may be parameterized with respect to a sequence of *formal input parameters* $\bar{x}$, denoted by $p[\bar{x}]$. The instantiation of $p[\bar{x}]$ with input values $\bar{v}$ of appropriate type, is denoted by $p[\bar{v}]$. In general, an update rule may be *nondeterministic*, in which case it may yield several states from a given state and given inputs. Thus, an *update rule* $p[x_1, \ldots, x_n]$ denotes a relation $[\![p]\!] \subseteq \mathit{States} \times \mathit{Values}^n \times \mathit{States}$. When p is deterministic, we consider $[\![p]\!]$ as a function $[\![p]\!] : \mathit{States} \times \mathit{Values}^n \rightarrow \mathit{States}$ and we say that the *invocation* (or *execution*) of $p[\bar{v}]$ from state s yields the state $[\![p]\!](s, \bar{v})$.

A *guard* φ is a state dependent formula that may contain free logic variables $\bar{x} = x_1, \ldots, x_n$, denoted by $\varphi[\bar{x}]$; φ is *closed* if it contains no free variables. Given values $\bar{v} = v_1 \ldots, v_n$ we write $\varphi[\bar{v}]$ for the replacement of x_i in φ by v_i for $1 \leq i \leq n$. A closed formula φ has the standard truth interpretation $s \models \varphi$ in a state s. A *guarded update rule* is a pair (φ, p) containing a guard $\varphi[\bar{x}]$ and an update rule $p[\bar{x}]$; intuitively (φ, p) limits the execution of p to those states and arguments $\bar{v}$ where $\varphi[\bar{v}]$ holds.

Definition 1. A *model program* P has the following components.

- A state space *States*.
- A value space *Values*.
- An *initial state* $s_0 \in$ *States*,
- A finite vocabulary Σ of *action symbols* partitioned into two disjoint sets
  - Σ^{c} of *controllable* action symbols, and
  - Σ^{o} of *observable* action symbols.
- A *reset* action symbol *Reset* $\in \Sigma^{\mathrm{c}}$.

– A family $(\varphi_f, p_f)_{f\in\Sigma}$ of guarded update rules.
  - The *arity* of f is the number of input parameters of p_f.
  - The arity of *Reset* is 0 and $[\![p_{\mathit{Reset}}]\!](s) = s_0$ for all $s \models \varphi_{\mathit{Reset}}$.

P is *deterministic* if, for all action symbols $f \in \Sigma$, p_f is deterministic.

An n-ary action symbol has logically the term interpretation, i.e. two ground terms whose function symbols are action symbols are equal if and only if the action symbols are identical and their corresponding arguments are equal. An *action* has the form $f(v_1, \ldots, v_n)$ where f is an n-ary action symbol and each v_i is a value that matches the required type of the corresponding input parameter of p_f. We say that an action $f(\bar{v})$ is *enabled* in a state s if $s \models \varphi(\bar{v})$. Notice the two special cases regarding reset: one when reset is always disabled ($\varphi_{\mathit{Reset}} = \mathit{false}$), in which case the definition of p_{Reset} is irrelevant, and the other one when reset is always enabled ($\varphi_{\mathit{Reset}} = \mathit{true}$), in which case p_{Reset} must be able to reestablish the initial state from any other program state.

We sometimes use *action* to mean an action symbol, when this is clear from the context or when the action symbol is nullary in which case there is no distinction between the two.

2.2 Example: *Recycling Robot*

We show a model program of a collection of *recycling robots* written in C# in Figure 1. A robot is a movable recycle-bin, it can either

1. move and *search* for a can if its power level (measured in percentage) is above the given threshold 30%, or
2. remain stationary and *wait* for people to dispose of a can if its power level is below the given threshold 50%.

Notice that both cases are possible when the power level is between 30% and 50%. A robot gets a reward by collecting cans. The reward is bigger when searching than while waiting, but each search reduces the power level of the robot by 30%. A robot can be *recharged* when it is not fully charged, i.e when the power level is less than 100%. New robots can be *started* dynamically provided that the total number of robots does not exceed a limit (if such a limit is given).

Actions In this example, the action symbols are `Start`, `Search`, `Wait` and `Recharge`, where the first three symbols are classified as being controllable and the last one is classified as being observable. All of the symbols are unary (i.e., they take one input). All actions have the form $f(i)$ where f is one of the four action symbols and i is a non-negative integer representing the id of a robot. The reset action is in this example implicit, and is assumed to be enabled in all states.

States The state signature has three state variables, a map `Robot.Instances` from object ids (natural numbers) to robots (objects of type `Robot`), and two field value maps `power` and `reward` that map robots to their corresponding power and reward values. The initial state is the state where all those maps are empty.

```
class Robot : EnumeratedInstance // The base class keeps track of created robot instances
{
    int power  = 0;
    int reward = 0;
}

class RobotModel
{
    static int maxNoOfRobots = ...;

    [Action]
    static void Start(int robotId)
    {
        Assume.IsTrue(Robot.Instances.Count < maxNoOfRobots &&
                      ¬ Robot.Instances.Count == robotId));
        new Robot(robotId);
    }

    [Action]
    static void Search(int robotId)
    {
        Assume.IsTrue(robotId ∈ Robot.Instances);
        Robot robot = Robot.Instances[robotId];
        Assume.IsTrue(robot.power > 30);

        robot.power  = robot.power - 30;
        robot.reward = robot.reward + 2;
    }

    [Action]
    static void Wait(int robotId)
    {
        Assume.IsTrue(robotId ∈ Robot.Instances);
        Robot robot = Robot.Instances[robotId];
        Assume.IsTrue(robot.power <= 50);

        robot.reward = robot.reward + 1;
    }

    [Action(Kind = Observable)]
    static void Recharge(int robotId)
    {
        Assume.IsTrue(robotId ∈ Robot.Instances);
        Robot robot = Robot.Instances[robotId];
        Assume.IsTrue(robot.power < 100);

        robot.power = 100;
    }
}
```

Fig. 1. Model Program of the *Recycling Robot* example.

Guarded update rules For each of the four actions f the guarded update rule (φ_f, p_f) is defined by the corresponding static method f of the `RobotModel` class. Given a robot id i and a state s, the guard $\varphi_f(i)$ is true in s, if all the `Assume.IsTrue` statements evaluate to *true* in s. Execution of $p_f[i]$ corresponds to the method invocation of $f(i)$. For example, in the initial state, say s_0, of the robot model, the single enabled action is `Start(0)`. In the resulting state $[\![p_{\mathtt{Start}}]\!](s_0, 0)$ a new robot with id 0 has been created whose reward and power are 0.

2.3 Deterministic model programs as interface automata

We use the notion of interface automata [8, 7] following the exposition in [7]. The view of a model program as an interface automaton is important for formalizing the conformance relation. To be consistent with the rest of the paper, we use the terms "controllable" and "observable" here instead of the terms "input" and "output" used in [7].

Definition 2. An *interface automaton* M has the following components:

- A set S of *states*.
- A nonempty subset S^{init} of S called the *initial states*.
- Mutually disjoint sets of *controllable actions* A^{c} and *observable actions* A^{o}.
- *Enabling functions* Γ^{c} and Γ^{o} from S to subsets of A^{c} and A^{o}, respectively.
- A *transition function* δ that maps a source state and an action enabled in the source state to a target state.

In order to identify a component of an interface automaton M, we index that component by M, unless M is clear from the context. Let P be a deterministic model program $(\mathit{States}, \mathit{Values}, s_0, \Sigma, \Sigma^{\text{c}}, \Sigma^{\text{o}}, \mathit{Reset}, (\varphi_f, p_f)_{f\in\Sigma})$. P has the following straightforward denotation $\llbracket P\rrbracket$ as an interface automaton:

$$\begin{aligned}
S_{\llbracket P\rrbracket} &= \mathit{States}\\
S^{\text{init}}_{\llbracket P\rrbracket} &= \{s_0\}\\
A^{\text{c}}_{\llbracket P\rrbracket} &= \{f(\bar{v}) \mid f \in \Sigma^{\text{c}}, \bar{v} \subseteq \mathit{Values}\}\\
A^{\text{o}}_{\llbracket P\rrbracket} &= \{f(\bar{v}) \mid f \in \Sigma^{\text{o}}, \bar{v} \subseteq \mathit{Values}\}\\
\Gamma^{\text{c}}_{\llbracket P\rrbracket}(s) &= \{f(\bar{v}) \in A^{\text{c}}_{\llbracket P\rrbracket} \mid s \models \varphi_f(\bar{v})\}\\
\Gamma^{\text{o}}_{\llbracket P\rrbracket}(s) &= \{f(\bar{v}) \in A^{\text{o}}_{\llbracket P\rrbracket} \mid s \models \varphi_f(\bar{v})\}\\
\delta_{\llbracket P\rrbracket}(s, f(\bar{v})) &= \llbracket P_f\rrbracket(s, \bar{v}) \quad (\text{for } f \in \Sigma,\ s \in \mathit{States},\ s \models \varphi_f(\bar{v}))
\end{aligned}$$

Note that $\delta_{\llbracket P\rrbracket}$ is well-defined, since P is deterministic. In light of the above definition we occasionally drop the distinction between P and the interface automaton $\llbracket P\rrbracket$ it denotes.

2.4 Implementing a model program as an interface automaton

A model program P exposes itself as an interface automaton through a *stepper* that provides a particular "walk" through the interface automaton one transition at a time. A stepper of P is implemented through the `IStepper` interface defined below. A stepper has an implicit *current state* that is initially the initial state of P. In the current state s of a stepper, the enabled actions are given by $\Gamma_{\llbracket P\rrbracket}(s)$. Doing a step in the current state s of the stepper according to a given action a corresponds to setting the current state of the stepper to $\delta_{\llbracket P\rrbracket}(s, a)$. The *Reset* action is handled separately and is not included in the set of currently enabled actions `EnabledControllables`.

```
interface IStepper
{
    Sequence<Action> EnabledControllables { get; }
    Sequence<Action> EnabledObservables { get; }
    void DoStep(Action action);

    void Reset();
    bool ResetEnabled { get; }
}
```

For conformance testing, an implementation is also assumed to be an interface automaton that is exposed through a stepper. If both the model and the IUT are interface automata with a common action signature, we test the conformance of the two automata using the refinement relation between interface automata as defined in [7].

3 Online testing algorithm

In this section we describe an algorithm that uses reinforcement learning to choose controllable actions during conformance testing of an implementation I against a model (specification) M. Both M and I are assumed to be given as model programs that expose an `IStepper` interface to the algorithm. In addition, the model exposes an interface that provides an abstract value of the current state of the model and an abstract value of any action enabled in a given state. It is convenient to view this interface as an extension `IModelStepper` of the IStepper interface:

```
interface IModelStepper : IStepper
{
    IComparable GetAbstractState(Action action);
    IComparable GetAbstractAction(Action action);
}
```

The main motivation for these functions is to divide the state space and the action space into equivalence classes that reflect “interesting” groups of states and actions for the purposes of coverage.

Example 1. Consider the Robot model. We could define the abstract states and abstract actions to be the concrete states and the concrete actions as follows. In other words, there is no grouping of either states or actions in this case.

```
class RobotModel : IModelStepper
{
    Sequence<Pair<int,int>> GetAbstractState(Action action)
    {
        return [(r.power, r.reward) | r in Robot.Instances]
    }
    Action GetAbstractAction(Action action);
    {
        return action;
    }
}
```

Example 2. A more interesting case is if we abstract away the id of the robot and project the state to the state of the robot doing the action, or a default value if the robot has not been started yet. This is reasonable because the robots do not interact with each other.

```
class RobotModel : IModelStepper
{
    Pair<int,int> GetAbstractState(Action action)
    {
        if (action.Name == "Start") return (-1, -1);
        Robot r = Robot.Instances[action.Argument(0)];
        return (r.power, r.reward);
    }
    string GetAbstractAction(Action action);
    {
        return action.Name;
    }
}
```

We use pseudo code that is similar to the original implementation code written in C# to describe the algorithm. We consider two controllable action selection *strategies* `Lct` and `Rlc` that are explained below, in addition to a memoryless purely randomized strategy `Rnd`.

```
enum Strategy {Rnd, Lct, Rlc}
```

The algorithm uses also an "oracle" to ask advice about whether to observe an observable action from the implementation, to call a controllable action, or to end a particular test run, during a single step of the algorithm. The oracle makes a random choice between controlling an observing when an observable action is enabled in the implementation at the same time as a controllable action is enabled in the model. If there are no observable actions enabled in the implementation and no controllable actions enabled in the model then the only meaningful advice the oracle can give is to end the current test run.

```
enum Advice {Control, Observe, End}

class Oracle
{
    IStepper M;
    IStepper I;

    Advice Advise()
    {
        bool noCtlrs = M.EnabledControllables.IsEmpty;
        bool noObs = I.EnabledObservables.IsEmpty;

        if (noCtlrs ∧ noObs) return Advice.End;
        if (noCtlrs) return Advice.Observe;
        if (noObs) return Advice.Control;
        return new Choose(Advice.Control, Advice.Observe);
    }
}
```

3.1 Top level loop

The top level loop of the algorithm is described by the following pseudo code.

```
class OnlineTesting
{
    IModelStepper M;
    IStepper I;
    int maxRun;
    int maxStep;
    Strategy h;
    Oracle oracle;

    bool ResetEnabled {get return M.ResetEnabled ∧ I.ResetEnabled;}

    void Run()
    {
        int run = 0;
        while (run < maxRun)
        {
            RunTestCase();                 // The core algorithm
            if (¬ResetEnabled) return;  // Cannot continue, must abort
            Reset();
            run += 1;
        }
    }
}
```

The inputs to the algorithm are a model program `M` that provides the `IModelStepper` interface and is the specification, a model program `I` that provides the `IStepper` interface an is the implementation under test, an upper bound `maxRun` on the total number of runs, an upper bound `maxStep` on the total number of steps (state transitions) per one run, a strategy `h`, and an oracle `oracle` as explained above.

3.2 The core algorithm

The algorithm keeps track of the *weights* of *abstract transitions* that have occurred during the test runs. An abstract transition is a pair (s, a) where s is an abstract state and a is an abstract action. The weight of an abstract transition is total number of times it has occurred plus one, since the algorithm was started. The abstract state and action values are calculated using the `IModelStepper` interface introduced above. This weight information is stored in a *test trace graph* that is updated dynamically and is initially empty.

```
class TestTraceGraph
{
    Map<AbstractTransition, int> F = ∅;   // Frequencies of explored abstract transitions
    IModelStepper M;

    int W(Action a)                       // Weights are positive
    {
        AbstractState s = M.GetAbstractState(a);
        AbstractAction b = M.GetAbstractAction(a);
        if ((s,b) ∈ F) return F[(s,b)]; else return 1;
    }

    void Update(Action a, int w)
    {
        AbstractState s = M.GetAbstractState(a);
        AbstractAction b = M.GetAbstractAction(a);
        F[(s,b)] = W(a) + w;
    }
}
```

The next controllable action is chosen by the algorithm from a nonempty set of controllable actions that are currently enabled, using the given strategy.

```
class TestTraceGraph
{
    Action ChooseAction(Sequence<Action> acts, Strategy h)
    {
        switch (h)
         {
            case Strategy.Lct:
                Action a = acts.Head;
                Pair<Set<Action>,int> lct =
                    acts.Tail.Reduce(Reducer,({acts.Head},W(acts.Head)));
                return lct.First.Choose();

            case Strategy.Rlc:
                Sequence<int> costs = [W(a) | a ∈ acts];
                int prod = ...; // Compute an approximate common multiple of costs
                Sequence<int> occurs = [prod/x | x ∈ costs];
                Bag<Action> bg = ⦃(acts[i], occurs[i]) | i < acts.Count⦄;
                return bg.Choose();

            default:
                return acts.Choose();
         }
     }
    Pair<Set<Action>,int> Reducer(Action a, Pair<Set<Action>,int> lct)
    {
        if (W(a) < lct.Second) return ({a}, w);
        else if (W(a) == lct.Second) return (lct.First ∪ {a}, w);
        else return lct;
    }
}
```

Lct: Choose an action that has the “least cost”. Here *cost* of an action a is measured as the current weight of the abstract transition (s, b), where s is the abstract state computed in the current model state with respect to a, and b is the abstract action corresponding to a, computed in the current model state. If several actions have the same least cost, one is chosen randomly from among those.

Rlc: Choose an action with a likelihood that is inversely proportional to its current cost, with cost having the same meaning as above. Intuitively this means that the least frequent actions are the most favored ones. In other words, if the candidate actions are $(a_i)_{i<k}$ for some k, having costs $(c_i)_{i<k}$, then the probability of selecting the action a_i is $c_i^{-1}/\sum_{j\neq i} c_j^{-1}$. The implementation uses a built-in bag construct to make such a choice.

Rnd: Make a random choice.

The algorithm runs one test case until, either a conformance failure occurs (in form of a violation of the refinement relation between $[\![\mathtt{M}]\!]$ and $[\![\mathtt{I}]\!]$), or until the given maximum number of steps has been reached.

```
class OnlineTesting
{
    TestTraceGraph ttg = new TestTraceGraph(M);

    bool RunTestCase()
    {
        int step = 0;
        while (step < maxStep)
        {
            Advice advice = oracle.Advise();

            if (advice == Advice.Control)
            {
                Sequence<Action> cs = M.EnabledControllables;
                Action c = ttg.ChooseAction(cs, h);
                ttg.Update(c, 1);                   // Increase the weight by 1
                M.DoStep(c);                        // Do the step in M

                if (c ∈ I.EnabledControllables)
                    I.DoStep(c);                    // Do the corresponding step in I
                else
                    return false;                   // Conformance failure occurred
            }
            else if (advice == Advice.Observe)
            {
                Sequence<Action> os = I.EnabledObservables;
                // This is an abstract view of the execution of the implementation, in reality
                // the implementation performs the choice itself and notifies the test harness
                Action o = os.Choose();
                I.DoStep(o);

                if (o ∈ M.EnabledObservables)
                {
                    ttg.Update(o, 1);               // Increase the weight by 1
                    M.DoStep(o);                    // Do the corresponding step in M
                }
                else
                    return false;                   // Conformance failure occurred

                #endregion
            }
            else
                return true;                        // No more steps can be performed
            step += 1;
        }
        return true;                                // The test case succeeds
    }
}
```

The *Lct* strategy is a greedy approach; it is very simple and relatively cheap to compute. However, it favors actions that have been used less frequently, and thus may systematically avoid long sequences of the same action, as is illustrated next.

Example 3. Consider a bounded stack of size n. The stack has two controllable actions, *top* and *push*, enabled in every state. The greedy strategy will alternate between these two actions until the stack is full. If we want to test the behavior of *push* when the stack is full, we need to continue testing for at least $2n$ steps (so that *push* is executed n times).

In the given algorithm, the weight increase is always 1. This value can be made domain specific and can vary depending both on the action and the current state, for example by extending the `IModelStepper` interface with a function that provides the wait increase for the given action in the current state and using that function instead of 1.

By using `Rlc`, the probability of selecting an action is inversely proportional to its frequency. Thus, the more an action has been selected the less likely it is that it will be selected again. So the potential problem shown in Example 3 is still there but ameliorated, since no enabled action is excluded from the choice.

4 Experiments

We used the Robot model to conduct a few experiments with the algorithm in order to evaluate and compare the different strategies. The main purpose was to see if the two proposed strategies `Lct` or `Rlc` are useful by providing better or at least as good coverage of the state space as the purely random approach. Since we are interested in state and transition coverage only, we ran the algorithm against a correct implementation. We ran the algorithm with a different maximum number of robots, different abstraction functions introduced in the examples above, and different limits on the total number of runs and the total number of steps per run. The experiments are summarized in Tables 1 and 2. We ran each case independently 50 times, the entries in the tables are shown on the form $m \pm \sigma$ where m is the mean of the obtained results and σ is the standard deviation. The absolute running times are shown only for comparison, the concrete machine was a 3GHz Pentium 4.

If states and actions are not grouped at all, by assuming the definitions given in Example 1, the majority of abstract transitions will occur only a single time and the strategies perform more or less as the random case, which is shown in Table 1. One can see that `Lct` performs marginally better than `Rnd` when the number of robots and the number of runs increases.

When the states and the actions are mapped to abstract values, as defined in Example 2, then `Lct` starts finding many more abstract states than `Rnd` as the number of robots grows. The robot id is ignored by the abstraction and thus concrete transitions of different robots that differ only by the id are mapped to the same abstract transition. Overall this will have the effect that the `Lct` approach will favor actions that transition to new abstract states. The same is true for the `Rlc` case but the increase in coverage is smaller.

Table 1. Execution of the online algorithm on the Robot model without grouping.

Parameters			Lct		Rlc		Rnd	
Robots	**Runs**	**Steps**	#**States**	t(ms)	#**States**	t(ms)	#**States**	t(ms)
1	1	100	100 ± 0	3	100 ± 0	1	100 ± 0	1
1	10	100	420 ± 11	20	415 ± 8	19	414 ± 9	15
1	100	100	503 ± 3	275	503 ± 3	241	502 ± 2	172
1	100	500	2485 ± 5	1303	2485 ± 5	1292	2485 ± 6	968
2	1	100	100 ± 0	3	100 ± 0	1	100 ± 0	2
2	10	100	951 ± 8	24	941 ± 10	22	938 ± 12	14
2	100	100	7449 ± 83	286	7085 ± 110	284	7055 ± 114	201
2	100	500	44119 ± 225	1548	42437 ± 339	1479	42364 ± 289	1040
5	1	100	100 ± 0	5	100 ± 0	3	100 ± 0	1
5	10	100	972 ± 3	42	971 ± 3	37	969 ± 4	18
5	100	100	9368 ± 17	516	9328 ± 22	468	9322 ± 24	297
5	100	500	49364 ± 19	2794	49330 ± 25	2541	49320 ± 19	1587

Table 2. Execution of the online algorithm on the Robot model with state grouping and action grouping.

Parameters			Lct		Rlc		Rnd	
Robots	**Runs**	**Steps**	#**States**	t(ms)	#**States**	t(ms)	#**States**	t(ms)
1	1	100	100 ± 0	3	100 ± 0	<1	100 ± 0	<1
1	10	100	417 ± 9	9	413 ± 8	7	416 ± 8	4
1	100	100	502 ± 2	100	503 ± 3	88	502 ± 2	44
1	100	500	2486 ± 5	508	2486 ± 6	417	2484 ± 6	234
2	1	100	100 ± 0	1	90 ± 3	<1	93 ± 5	<1
2	10	100	419 ± 7	10	284 ± 21	9	237 ± 8	4
2	100	100	502 ± 3	106	437 ± 12	96	293 ± 6	46
2	100	500	2485 ± 5	561	1602 ± 33	506	1324 ± 15	241
5	1	100	100 ± 0	<1	66 ± 4	1	61 ± 2	<1
5	10	100	418 ± 10	10	279 ± 30	11	117 ± 5	5
5	100	100	503 ± 3	115	472 ± 7	116	155 ± 7	50
5	100	500	2484 ± 5	561	1696 ± 96	657	582 ± 10	247
5	100	1000	4949 ± 8	1200	2467 ± 95	1388	1088 ± 13	540
10	10	100	418 ± 9	10	293 ± 25	12	91 ± 6	5
10	100	100	502 ± 3	103	473 ± 6	137	128 ± 6	59
10	100	1000	4951 ± 11	1131	3541 ± 198	1718	602 ± 10	578
10	1000	1000	4985 ± 8	12521	4352 ± 66	18043	654 ± 9	5953

The Robot case study is representative for models that deal with multiple agents at the same time, which is a typical case in testing of multi-threaded software [20]. Often the threads are mostly independent, an abstraction technique that can be used in this context is to look at the part of the state that belongs to the agent doing the action. This is an instance of so-called multiple state-grouping approach that is also used as an exploration technique for FSM generation [4]. This is exactly what is done in Example 2. It seems that `Lct` is a promising heuristic for online testing of these kinds of models. One can note that, the coverage provided by the random approach degrades almost by half as the number of robots is doubled (for example from 5 to 10).

5 Related work

The basic idea of online testing has been introduced in the context of labeled transition systems using ioco theory [3, 17, 19] and implemented in the TorX tool [18]. TGV [11] is another tool frequently used for online or on-the-fly test generation that uses ioco.

Ioco theory is a formal testing theory based on labeled transition systems with input actions and output actions. Interface automata [7] are suitable for the game view [5] of online testing and provide the foundation for the conformance relation that we use. Online testing with model programs in the SpecExplorer tool is discussed in in [20]. The algorithm in [20] does not use learning, and as far as we know learning algorithms have not been considered in the context of model based testing. The relation between ioco and refinement of interface automata is briefly discussed in [20]. Specifications given by a guarded command language are used also in [15].

In Black-box testing, some work [14] has been done which uses supervised learning procedures. As far as we know, no previous work has addressed online testing with learning in the context of Model Based Testing. The main intuition behind our algorithm is similar to an anti-ant approach [13] used for test case generation form UML diagrams. From the game point of view, the online testing problem is a $1\frac{1}{2}$-player game. It is known that $1\frac{1}{2}$-player games are Markov Decision Processes [6]. The view of finite explorations of model programs for offline test case generation as negative total reward Markov decision problems with infinite horizon are studied in [2].

6 Open problems and future work

One of the interesting areas that is also practically very relevant is to gain better understating of approaches for online testing that learn from model-coverage that uses abstractions. The experiments reported in Section 4 exploited that idea to a certain extent by using state and action abstraction through the `IModelStepper` interface, but the general technique and theory need to be developed further. Such abstraction functions can either be user-provided [9, 4] or automatically generated from program text similar to iterative refinement [15].

Currently we have an implementation of the presented algorithm using a modeling library developed in C#. As a short-term goal, we are working on a more detailed report where we are considering larger case studies.

The algorithm can also be adapted to run without a model, just as a semi-random (stress) testing tool of implementations. In that case the history of used actions is kept solely based on the test runs of the implementation. In this case, erroneous behaviors would for example manifest themselves through unexpected exceptions thrown by the implementation, rather than trough conformance violations.

Acknowledgment

We thank Nikolai Tillmann and Wolfgang Grieskamp for many valuable discussions and for help in using the underlying exploration framework XRT [10] during the initial implementation of the ideas. We thank Luca de Alfaro and Wolfram Schulte for valuable comments on earlier drafts of this paper.

References

1. R. Alur, C. Courcoubetis, and M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *Proc. 27th Ann. ACM Symp. Theory of Computing*, pages 363–372, 1995.
2. A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to test. Technical Report MSR-TR-2005-04, Microsoft Research, January 2005. Short version of this report was presented at FATES 2005.
3. E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In *Summer School MOVEP'2k – Modelling and Verification of Parallel Processes*, volume 2067 of *LNCS*, pages 187–193. Springer, 2001.
4. C. Campbell and M. Veanes. State exploration with multiple state groupings. In D. Beauquier, E. Börger, and A. Slissenko, editors, *12th International Workshop on Abstract State Machines, ASM'05, March 8–11, 2005, Laboratory of Algorithms, Complexity and Logic, University Paris 12 – Val de Marne, Créteil, France*, pages 119–130, 2005.
5. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Synchronous and bidirectional component interfaces. In *CAV*, pages 414–427, 2002.
6. K. Chatterjee, L. de Alfaro, and T. A. Henzinger. Trading memory for randomness. In *QEST*, pages 206–217, 2004.
7. L. de Alfaro. Game models for open systems. In N. Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 269 – 289. Springer, 2004.
8. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM, 2001.
9. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA'02*, volume 27 of *Software Engineering Notes*, pages 112–122. ACM, 2002.
10. W. Grieskamp, N. Tillmann, and W. Schulte. Xrt – exploring runtime for .NET architecture and applications. Technical Report MSR-TR-2005-63, Microsoft Research, June 2005. Presented at SoftMC 2005.
11. C. Jard and T. Jéron. TGV: theory, principles and algorithms. In *The Sixth World Conference on Integrated Design and Process Technology, IDPT'02*, Pasadena, California, June 2002.
12. L. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey, 1996.
13. H. Li and C. Lam. Using anti-ant-like agents to generate test threads from the uml diagrams. In *Proc. Testcom 2005*, LNCS. Springer, 2005.
14. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE*, pages 225–240, 1999.
15. C. S. Păsăreanu, R. Pelánek, and W. Visser. Concrete model checking with abstract matching and refinement. In *Computer Aided Verification (CAV 2005)*, volume 3576 of *LNCS*, pages 52–66. Springer, 2005.
16. R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT, 1998. URL: http://www.cs.ualberta.ca/ sutton/book/ebook/the-book.html.
17. J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.
18. J. Tretmans and E. Brinksma. TorX: Automated model based testing. In *1st European Conference on Model Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.

19. M. van der Bij, A. Rensink, and J. Tretmans. Compositional testing with ioco. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing: Third International Workshop, FATES 2003*, volume 2931 of *LNCS*, pages 86–100. Springer, 2004.
20. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 273–282. ACM, 2005.