Play to Test

Andreas Blass*[1], Yuri Gurevich[2], Lev Nachmanson[2], and Margus Veanes[2]

[1] University of Michigan, Ann Arbor, MI, USA
ablass@umich.edu
[2] Microsoft Research, Redmond, WA, USA
gurevich,levnach,margus@microsoft.com

Abstract. Testing tasks can be viewed (and organized!) as games against nature. We study reachability games in the context of testing. Such games are ubiquitous. A single industrial test suite may involve many instances of a reachability game. Hence the importance of optimal or near optimal strategies for reachability games. One can use linear programming or the value iteration method of Markov decision process theory to find optimal strategies. Both methods have been implemented in an industrial model-based testing tool, Spec Explorer, developed at Microsoft Research.

1 Introduction

If you think of playful activities, software testing may not be the first thing that comes to your mind, but it is useful to see software testing as a game that the tester plays with the implementation under test (IUT). We are not the first to see software testing as a game [2] but our experience with building testing tools at Microsoft leads us to a particular framework.

An industrial tester typically writes an elaborate test harness around the IUT and provides an application program interface (API) for the interaction with the IUT. You can think of the API sitting between the tester and the IUT. It is symmetric in the sense that it specifies the methods that the tester can use to influence IUT and the methods that the IUT can use to pass information to the tester. From tester's point of view, the first methods are *controllable actions* and the second methods are *observable actions*.

The full state of the IUT is hidden from the tester. Instead, the tester has a model of the IUT's behavior. A model state is given by the values of the model variables which can be changed by means of actions whether controllable or observable. But this is not the whole story. In addition, there is an implicit division of the states into *active* and *passive*; in other words there is an implicit Boolean state variable "the state is active". The initial state is active but, whenever the model makes a transition to a target state where an observable action is enabled, the target state is passive; the target state is active otherwise. At a passive state, the tester waits for an observable action. If nothing happens within a state-dependent timeout, the tester interprets the timeout itself as a default observable action which changes the passive state into an active state

* Much of the research reported here was done while the first author was a visiting researcher at Microsoft Research.

with the same values of the explicit variables. At an active state the tester applies one of the enabled controllable actions. Some active states are *final*; this is determined by a predicate on state variables. The tester has an option of finishing the game whenever the state is final.

We presume here that the model has already been checked for correctness. We are testing IUT for the conformance to the model. Here are some examples of how you detect nonconformance. Suppose that the model is in a passive state s. If only actions a, b are enabled in s but you observe an action c, different from a and b, then you witness a violation of the conformance relation. If the model tells you that any non-timeout action enabled in s returns a positive integer but the IUT throws an exception or returns -1, then, again, you have discovered a conformance violation. This kind of conformance relation is close to the one studied by de Alfaro [11].

In a given passive state the next observable action and its result are not determined uniquely in general. What are the possible sources of the apparent nondeterminism? One possible source is that the IUT interacts with the outside world in a way that is hidden from the tester. For example, it is in many cases not desirable for the tester to control the scheduling of the execution threads of a multithreaded IUT; it may be even impossible in the case of a distributed IUT. Another possible source of nondeterminism is that the model state is more abstract than the IUT state. For example, the model might use a set to represent a collection of elements that in reality is ordered in one way or another in the IUT.

The group of Foundations of Software Engineering at Microsoft Research developed a tool, called Spec Explorer, for writing, exploring, and validating software models and for model-based testing of software. Typically the model is more abstract and more compact than the IUT; nevertheless its state space can be infinite or very large. It is desirable to have a finite state space of a size that allows one to explore the state space. To this end, Spec Explorer enables the tester to generate a finite but representative set of parameters for the methods. Also, the tester can indicate a collection of predicates and other functions with finite (and typically small) domains and then follow only the values of these functions during the exploration of the model [13]. These and other ways of reducing the state space are part of a finite state machine (FSM) generation algorithm implemented in the Spec Explorer tool; the details fall outside the scope of this paper. The theoretical foundations of the tool are described in [8]. The tool is available from [1].

The game that we are describing is an example of so-called games against nature which is a classical area in optimization and decision making under uncertainty going back all the way to von Neumann [26]. Only one of the two players, namely the tester, has a goal. The other player is disinterested and makes random choices. Such games are also known as $1\frac{1}{2}$-player games. We make a common assumption that the random choices are made with respect to a known probability distribution. How do we know the probability distribution? In fact, we usually don't. Of course symmetry considerations are useful, but typically they are insufficient to determine the probability distribution. One approximates the probability distribution by experimentation.

The tester may have various goals. Typically they are cover-and-record goals e.g. visit every state (or every state-to-state transition) and record everything that happened

in the process. Here we study *reachability games* where the goal is to reach a final state. It is easy to imagine scenarios where a reachability game is of interest all by itself. But we are interested in reachability games primarily because they are important auxiliary games. In an industrial setting, the tester often runs test suites that consist of many test segments. The state where one test segment naturally ends may be inappropriate for starting the next segment because various shared resources have been acquired or because the state is littered with ancillary data. The shared resources should be freed and the state should be cleaned up before the segment is allowed to end. Final states are such clean states where a new segment can be initiated. And so the problem arises of arriving at one of the final states.

It is a priori possible that no final state is reachable from the natural end-state of a test segment. In such a case it would be impossible to continue a test suite. Spec Explorer avoids such unfortunate situations by pruning the FSM so that that it becomes *transient* in the following sense: from every state, at least one final state is reachable (unless IUT crashes). The pruning problem can be solved efficiently using a variation of [10, Algorithm 1] (which is currently implemented in Spec Explorer), or the improved algorithm in [9, Section 4].

The tester cannot run a vast number of test segments by hand. The testing activity at Microsoft gets more and more automated. The Spec Explorer tool plays an important role in the process. Now is the time to expose a simplification that we made above speaking about the tester making moves. It is a testing tool (TT) that makes moves. The tester programs a game strategy into the TT.

The reachability games are so ubiquitous that it is important to compute optimal or nearly optimal strategies for them. You compute a strategy once for a given game and then you use it over and over many times. Since reachability games are so important for us, we research them from different angles.

In Section 2, reachability games are formulated, analysed and solved by means of linear programming and various known results, in particular [10, Theorem 9]. We associate a state dependent cost with each action. The optimal strategy minimizes the expected total cost which is the sum of the costs incurred during the execution. We observe that a reachability game can be viewed as a negative Markov decision process with infinite horizon [22]; the stopping condition is the first arrival at a final state. This allows us, in Section 3, to solve any reachability problem using the well known value iteration method. Theorem 7.3.10 in [22] guarantees the convergence. In Section 4 we provide experimental results by applying both methods to typical model programs using the Spec Explorer tool. Section 5 is devoted to related work. Section 6 gives a short conclusion.

Often the value iteration method works faster than the simplex method, but linear programming has its advantages and sheds some more light on the problem. In general, the applicability of one method does not imply the applicability of the other. Spec Explorer makes use of both, linear programming and value iteration, to generate strategies. Recall that strategy generation happens upon completion of FSM generation and a possible elimination of states from which no final state is reachable. The step of getting from the model program to a particular test graph is illustrated with the following example.

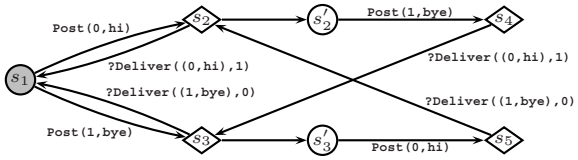

Fig. 1. Sample test graph generated by Spec Explorer from the chat model; diamonds represent passive states; ovals represent active states; links to s_2' and s_3' represent transitions to active mode; observable actions are prefixed by a question mark.

Example: Chat Session We illustrate here how to model a simple reactive system. This example is written in the AsmL specification language [16]. The chat session lets a client post messages for the other clients. The state of the system is given by the tuple `(clients,queue,recipients)`, where `clients` is the set of all clients of the session, `queue` is the queue of pending (sender,text) messages, and `recipients` is the set of remaining recipients of the first message in the queue called the *current* message.

```
var clients as Set of Integer
var queue as Seq of (Integer,String)
var recipients as Set of Integer
```

Posting a message is a *controllable* action. The action is enabled if the Boolean expression given by the `require` clause holds. Notice that the second conjunct of the enabling condition is trivially true if the queue is empty.

```
Post(sender as Integer, text as String)
  require sender in clients and
          forall msg in queue holds msg.First <> sender
  if queue.IsEmpty then recipients := clients - {sender}
  queue := queue + [(sender,text)]
```

Delivery of a message is an *observable* action. The current message must be delivered to all the clients other than the sender. Upon each delivery, the corresponding receiver is removed from the set of recipients. If there are no more recipients for the current message, the queue is popped and the next message (if any) becomes the current one. In other words, the specification prescribes that the current message must be delivered to all the recipients before the remainder of the queue is processed.

```
Deliver(msg as (Integer, String), recipient as Integer)
  require not queue.IsEmpty and then
          queue.Head = msg and recipient in recipients
  if recipients.Size = 1 then
    if queue.Length = 1 then recipients := {}
    else recipients := clients - {queue.Tail.Head.First}
    queue := queue.Tail
  else recipients := recipients - {recipient}
```

A good example of a natural finality condition in this case is `queue.IsEmpty`, specifying any state where there are no pending messages to be delivered.

If we configure the chat session example in Spec Explorer so that the initial state is $(\{0,1\},[],\{\})$ with two clients 0 and 1, where client 0 only posts “hi”, and client 1 only

posts “bye”, then we get the test graph illustrated in Figure 1. The initial state is s_1, and that is also the only final state with the above finality condition.

2 Reachability games and linear programming

We use a modification of the definition of a test graph in [20] to describe nondeterministic systems. A *test graph* G has a set V of *vertices* or *states* and a set E of directed *edges* or *transitions*. The set of states splits into three disjoint subsets: the set V^{a} of *active* states, the set V^{p} of *passive* states, and the set V^{g} of *goal* states. Without loss of generality, we may assume that V^{g} consists of a single goal state g such that no edge exits from g; the reduction to this special case is obvious.

There is a *probability function* p mapping edges exiting from passive nodes to positive real numbers such that, for every $u \in V^{\mathrm{p}}$,

$$\sum_{(u,v)\in E} p(u,v) = 1. \tag{1}$$

Notice that this implies that for every passive state there is at least one edge starting from it, and we assume the same for active states. Finally, there is a *cost function* c from edges to positive real numbers. One can think about the cost of an edge as, for example, the time for IUT to execute the corresponding function call. Formally, we denote by G the tuple $(V, E, V^{\mathrm{a}}, V^{\mathrm{p}}, g, p, c)$.

We assume also that for all $u, v \in V$ there is at most one edge from u to v. (This is not necessarily the case in applications; the appropriate reduction is given in Section 2.3.) Thus $E \subset V \times V$. It is convenient to extend the cost function to $V \times V$ by setting $c(u,v) = 0$ for all $(u,v) \notin E$.

2.1 Reachability game

Let $G = (V, E, V^{\mathrm{a}}, V^{\mathrm{p}}, g, p, c)$ be a test graph and u a vertex of it. The *reachability game* $R(u)$ over G is played by a testing tool (TT) and an implementation under test (IUT). The vertices of G are the states of $R(u)$, and u is the initial state. The current state of the game is indicated by a marker. Initially the marker is at u. If the current state v is active then TT moves the marker from v along one of graph edges. If the current state v is passive then IUT picks an edge (v,w) with probability $p(v,w)$ and moves the marker from v to w. TT wins if the marker reaches g. With every transition e the cost $c(e)$ is added to the total game cost.

A *strategy* for (the player TT in) G is a function S from V^{a} to V such that $(v, S(v)) \in E$ for every $v \in V^{\mathrm{a}}$. Let $R(u,S)$ be the sub-game of $R(u)$ when TT plays according to S.

We would like to evaluate strategies and compare them. To this end, for every strategy S, let $M_S[v]$ be the expected cost of the game $R(v,S)$. Of course, the expected cost may diverge, in which case we set $M_S[v] = \infty$. We say that M_S is *defined* if $M_S[v] < \infty$ for all v. If, for example, c reflects the durations of transition executions

then M_S reflects the expected game duration. The expected cost function satisfies the following equations.

$$\begin{aligned} M_S[g] &= 0 \\ M_S[u] &= c(u, S(u)) + M_S[S(u)] \quad \text{for } u \in V^{\mathrm{a}} \\ M_S[u] &= \sum_{(u,v)\in E} \{p(u,v)(c(u,v) + M_S[v])\} \quad \text{for } u \in V^{\mathrm{p}} \end{aligned} \tag{2}$$

We call a strategy S *optimal* if $M_S[v] \leq M_{S'}[v]$ for every strategy S' and every $v \in V$, or, more concisely, if $M_S \leq M_{S'}$ for every strategy S'. How can we construct an optimal strategy? Our plan is to show that the cost vector M of an optimal strategy is an optimal solution of a certain linear programming problem. This will allow us to find such an M. Then we will define a strategy S such that, for all active states u,

$$c(u, S(u)) + M[S(u)] = \min_{(u,v)\in E} \{c(u,v) + M[v]\}. \tag{3}$$

We will define transient test graph and prove that the strategy S is optimal when the test graph is transient.

Let us suppose from here on that the set V of states is $\{0, 1, ..., n-1\}$ and that the goal state $g = 0$. Consider a strategy S over G. We denote by P_S the following $n \times n$ matrix of non-negative reals:

$$P_S[u,v] = \begin{cases} p(u,v), & \text{if } u \in V^{\mathrm{p}} \text{ and } (u,v) \in E\ ; \\ 1, & \text{if } u \in V^{\mathrm{a}} \text{ and } v = S(u) \text{ or if } u = v = 0; \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

$P_S[u, v]$ is the probability of the move (u, v) when the game $R(u, S)$ is in state u, except that there is no move from state 0. (We could have added an edge $(0, 0)$ in which case there would be no exception.) So P_S is a *probability matrix* (also called a *stochastic matrix*) [12] since all entries are nonnegative and each row sum equals 1.

A strategy S is called *reasonable* if for every $v \in V$ there exists a number k such the probability to reach the goal state within at most k steps in the game $R(v, S)$ is positive. Intuitively, a reasonable strategy may be not optimal but eventually it has some chance of leading the player to the goal state.

Lemma 1. *A strategy S is reasonable for a test graph G if and only if, for some k, there exists, for each vertex v, a path P_v of length at most k from v to the goal state such that, whenever an active vertex w occurs in P_v, then the next vertex in P_v is $S(w)$.*

Proof. The "only if" half is obvious, because a play in $R(v, S)$ that reaches the goal state in at most k steps traces out a path P_v of the required sort. For the "if" half, recall that all the edges of G have positive probabilities. Thus, P_v has a positive probability of being traced by a play of $R(v, S)$, and so this game has a positive probability of reaching g in at most k steps. □

A nonempty subset U of V is *closed* if the game never leaves U after starting at any vertex in U. If S is reasonable, no subset U of $V - \{g\}$ is closed under the game $R(u, S)$, for any $u \in U$. This property is used to establish the following facts.

We let P'_S denote the *minor* of P_S obtained by crossing out from P_S row 0 and column 0. The following lemma follows from [12, Proposition M.3].

Lemma 2. *Let S be a reasonable strategy. Then*

$$\lim_{k\to\infty} {P'_S}^k = 0; \tag{5}$$

$$\sum_{k=0}^{\infty} {P'_S}^k = (I - P'_S)^{-1}. \tag{6}$$

Reasonable strategies can be characterized in terms of their cost vectors as follows. Complete proofs are given in [7].

Lemma 3. *A strategy S is reasonable if and only if M_S is defined. Moreover, if M_S is defined then $M'_S = (I - P'_S)^{-1} b'_S$ where M'_S and b'_S are the projections to the set $V - \{0\}$ of the expected cost vector M_S and the "immediate cost" vector b_S defined by*

$$b_S[u] = \sum_{v\in V} P_S[u,v]c(u,v) \quad (\forall u \in V).$$

A vertex v of a test graph is called *transient* if the goal state is reachable from v. We say that a test graph is *transient* if all its non-goal vertices are transient. There is a close connection between transient graphs and reasonable strategies.

Lemma 4. *A test graph is transient if and only if it has a reasonable strategy.*

In practice, the probabilities and costs in a test graph may not be known exactly. It is therefore important to know that, as long as the graph is transient, the optimal cost is robust, in the sense that it is not wildly sensitive to small changes in the probabilities and costs. This sort of robustness is, of course, just continuity, which the next lemma establishes.

Lemma 5. *For transient test graphs, the optimal cost vector M is a continuous function of the costs $c(u,v)$ and the probabilities $p(u,v)$.*

Proof. Throughout this proof, "continuous" means as a function of the costs $c(u,v)$ and the probabilities $p(u,v)$.

Temporarily consider any fixed, reasonable strategy S for the given test graph. Thanks to Lemma 1, S remains reasonable when we modify the probabilities (and costs) as long as they remain positive.

The formula for b_S in Lemma 3 shows that this vector is continuous. So is the matrix $I - P'_S$. Since the entries in the inverse of a matrix are, by Cramer's rule, rational functions of the entries of the matrix itself, we can infer the continuity of $(I - P'_S)^{-1}$ and therefore, by Lemma 3, the continuity of M'_S. Since the only component of M_S that isn't in M'_S is 0, we have shown that M_S is continuous.

Now un-fix S. The optimal cost vector M is simply the component-wise minimum of the M_S, as S ranges over the finite set of reasonable strategies. Since the minimum of finitely many continuous, real-valued functions is continuous, the proof of the lemma is complete. □

Of course, we cannot expect the optimal strategy to be a continuous function of the costs and probabilities. A continuous function from the connected space of cost-and-probability functions to the finite space of strategies would be constant, and we certainly cannot expect a single strategy to be optimal independently of the costs and probabilities. Nevertheless, the optimal strategies are robust in the following sense.

Suppose S is optimal for a given test graph, and let an arbitrary $\varepsilon > 0$ be given. Then after any sufficiently small modification of the costs and probabilities, S will still be within ε of optimal. Indeed, the continuity, established in the proof of Lemma 5, of the function M_S and of its competitors $M_{S'}$ arising from other strategies, ensures that, if we modify the costs and probabilities by a sufficiently small amount, then no component M_S will increase by more than $\varepsilon/2$ and no component of any $M_{S'}$ will decrease by more than $\varepsilon/2$. Since $M_S \leq M_{S'}$ before the modification, it follows that $M_S \leq M_{S'} + \varepsilon$ afterward.

A similar argument shows that, if S is strictly optimal for a test graph G, in the sense that any other S' has all components of $M_{S'}$ strictly larger than the corresponding components of M_S, then S remains strictly optimal when the costs and probabilities are modified sufficiently slightly. Just apply the argument above, with ε smaller than the minimum difference between corresponding components of M_S and any $M_{S'}$.

2.2 Linear programming

Ultimately, our goal is to compute optimal strategies for a given test graph G. We start by formulating the properties of the expected cost vector M as the following optimization problem. Let $\mathbf{d}$ be the constant row vector $(1, ..., 1)$ of length $|V| = n$.

LP: Maximize $\mathbf{d}M$, i.e. $\sum_{u \in V} M[u]$, subject to $M \geq 0$ and

$$\begin{cases} M[0] \leq 0 \\ M[u] \leq c(u,v) + M[v] \quad \text{for } u \in V^{\mathrm{a}} \text{ and } (u,v) \in E \\ M[u] \leq \sum_{(u,v) \in E} \{p(u,v)(c(u,v) + M[v])\} \quad \text{for } u \in V^{\mathrm{p}} \end{cases}$$

Intuitively, a feasible solution M to LP, e.g. $M = \mathbf{0}$, approximates an expected cost vector from below. Assuming that an optimal solution exists, the larger the value of $\mathbf{d}M$ is for a feasible solution M, the closer M is to an optimal cost vector.

Test graphs reduce to a subclass of negative stationary Markov decision processes (MDPs) with an infinite horizon, where rewards are negative and thus regarded as costs, strategies are stationary, i.e. time independent, and there is no finite upper bound on the number of steps in the process. A transient test graph reduces to a negative MDP where the probability to reach the goal from every state is positive. The optimization criterion for our strategies corresponds to the expected *total* reward criterion, rather than the expected *discounted* reward criterion used in discounted Markov decision problems. The total reward optimization problem is in general harder than the discounted reward optimization problem. However, for this subclass of negative MDPs the total reward optimization problem is known to be solvable by linear programming and yields a unique optimal solution [10, Theorem 9]. From Alfaro's Theorem 9 [10] we get the following corollary for test graphs. A careful self-contained proof of the corollary is given in [7].

Corollary 1. *The following statements are equivalent for all test graphs G.*

(a) G is transient.
(b) G has a reasonable strategy.
(c) LP for G has a unique optimal solution M. Moreover, $M = M_S$ for some strategy S and the strategy S is optimal.

Now we presume that the test graph is transient and show how to construct an optimal strategy. By applying Corollary 1 and solving LP, find the cost vector M of some optimal strategy O. In our notation, $M = M_O$. Construct strategy S so that equation (3) is satisfied for every active state u.

Proposition 1. *The constructed strategy S is optimal.*

Notice that, even though an optimal strategy S yields a unique cost vector M_S, S itself is not necessarily unique. Consider for example a test graph without passive states and with edges $\{1 \xrightarrow{1} 2,\ 2 \xrightarrow{10} 0,\ 1 \xrightarrow{10} 3,\ 3 \xrightarrow{1} 0\}$ that are annotated with costs; clearly both of the two possible strategies are optimal.

2.3 Graph transformation

We made the assumption that for each two vertices in the graph there is at most one edge connecting them. Let us show that we did not lose any generality by assuming this. For an active state u and for any $v \in V$ let us choose an edge leading from u to v with the smallest cost and discard all the other edges between u and v. For a passive state u replace the set of multiple edges D between u and v with one edge e such that $p(e) = \sum_{e' \in D} p(e')$ and $c(e) = (\sum_{e' \in D} p(e')c(e'))/p(e)$. This merging of multiple edges into a single edge does not change the expected cost of one step from u. The graph modifications have the following impact on LP. With removal of the edges exiting from active states we drop the corresponding redundant inequalities. The introduction of one edge for a passive state with changed c and p functions does not change the coefficients before $M[v]$ in LP in the inequality corresponding to passive states and therefore does not change the solution of LP.

2.4 Graph compression

Every test graph is equivalent, as far as our optimization problems are concerned, to one in which no edge joins two passive vertices. The idea is to replace the edges leaving a passive vertex u in the following manner. Consider all paths emanating from u, passing through only passive vertices, but then ending at an active vertex or the goal vertex. Each such path has a probability, obtained by multiplying the probabilities of its edges, and it has a cost, obtained by adding the costs of its edges. Replace each such path by a single edge, from u to the final, active vertex in the path; give this new edge the same probability and cost that the path had. If this replacement process produces several edges joining the same pair of vertices, transform them to a single edge as in Subsection 2.3. The details of test graph compression are given in [7].

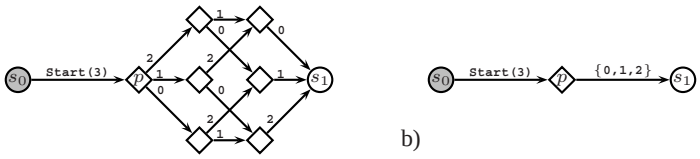

Fig. 2. a) Test subgraph obtained by exploring the extended chat model with 3 clients up to the posting phase; transitions from passive states (diamonds) are labeled by the respective client entering the session. b) Same subgraph after compression.

One may wonder if such a compression is worthwhile. The answer depends to a great extent on the topology of the test graph. It may sometimes pay off to apply the compression to certain subgraphs of the full test graph, rather than to the whole test graph. Let us illustrate a fairly common situation that arises in testing highly concurrent systems where the compression would reduce the number of states and edges. We revisit the chat model above and extend it as follows. There is an additional state variable `nClients` representing the number of clients entering in the chat session, so the state is given by the tuple `(nClients,clients,queue,recipients)`. There is a new *controllable* action `Start` that starts the entering phase of clients by updating `nClients`. There is also a new *observable* action `Enter` representing the event of a client entering the session. A client that has already entered the session cannot enter it again.

```
var nClients as Integer
Start(n as Integer)
  require nClients = 0 and n > 0
  nClients := n
Enter(c as Integer)
  require nClients > 0 and c in {0..nClients-1} - clients
  clients := clients + {c}
```

Assume also that the enabling condition (`require` clause) of the `Post` action is extended with the condition that the entering phase was started and that all clients have entered the session, i.e., `nClients > 0 and clients.Size = nClients`. So the "posting" phase is not started until all clients have entered the session. Suppose that the initial state s_0 is $(0, \emptyset, [], \emptyset)$. By generating the FSM from the model program with 3 clients, the initial part of the test graph up to the posting phase that starts in state s_1 is illustrated in Figure 2.a. The compression of the subgraph between the states s_0 and s_1 would yield the subgraph shown in Figure 2.b with a single passive state p and a transition from p to s_1 representing the composed event of all three clients having entered the session in some order.

The effect of the compression algorithm is in some cases, such as in this example, similar to partial order reduction. Obviously, reducing the size of the test graph improves feasibility of the linear programming approach. However, for large graphs we use the value iteration algorithm, described next. Due to the effectiveness of value iteration the immediate payoff of compression is not so clear, unless compression is simple and the number of states is reduced by an order of magnitude. We are still investigating the practicality of compression and it is not yet implemented in the Spec Explorer tool.

3 Value iteration

Value iteration is the most widely used algorithm for solving discounted Markov decision problems (see e.g. [22]). Reachability games give rise to non-discounted Markov decision problems. Nevertheless the value iteration algorithm applies; this is a practical approach for computing strategies for transient test graphs.

Let $G = (V, E, V^{\mathrm{a}}, V^{\mathrm{p}}, g, p, c)$ be a test graph. The classical value iteration algorithm works as follows on G.

Value iteration Let $n = 0$ and let M^0 be the zero vector with coordinates V so that every $M^0[u] = 0$. Given n and M^n, we compute M^{n+1} (and then increment n):

$$M^{n+1}[u] = \begin{cases} \min_{(u,v)\in E}\{c(u,v) + M^n[v]\}, & \text{if } u \in V^{\mathrm{a}}; \\ \sum_{(u,v)\in E} p(u,v)(c(u,v) + M^n[v]), & \text{if } u \in V^{\mathrm{p}}; \\ 0, & \text{if } u = 0. \end{cases} \tag{7}$$

Value iteration for negative MDPs with the expected total reward criterion, or *negative Markov decision problems* for short, does not in general converge to an optimal solution, even if one exists. However, if there exists a strategy for which the expected cost is finite for all states [22, Assumption 7.3.1], then value iteration *does* converge for negative Markov decision problems, see [22, Theorem 7.3.10] or [10, Theorem 10]. In light of lemmas 3 and 4, this implies that value iteration converges for transient test graphs. Let us make this more precise, as a corollary of Theorem 7.3.10 in [22] or Theorem 10 in [10].

Corollary 2. *Let G be a transient test graph as above. For any $\varepsilon > 0$, there exists N such that, for all $n \geq N$ and all states $u \in V$, $M^*[u] - M^n[u] < \varepsilon$, where M^* is the optimal cost vector.*

The iterative process, generally speaking, does not reach a fixed point in finitely many iterations. Consider the test graph in Figure 3. It is not difficult to calculate that

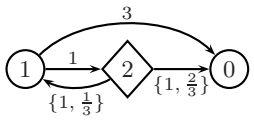

Fig. 3. Sample test graph; transitions from active states are labeled by their costs; transitions from passive states are labeled by their costs and probabilities.

the infinite sequence $(M^n[1])_{n=1}^{\infty}$ computed by (7) is

$$1,\ 2,\ 2\tfrac{1}{3}\ 2\tfrac{2}{3},\ 2\tfrac{7}{9},\ 2\tfrac{8}{9},\ 2\tfrac{25}{27},\ 2\tfrac{26}{27},\ \ldots,\ 2\tfrac{3^i-2}{3^i},\ 2\tfrac{3^i-1}{3^i},\ \ldots$$

that converges to $M^*[1] = 3$.

When should we terminate the iteration? Given a cost vector M let S_M denote any strategy defined so that equation (3) is satisfied for every active state u. Further, let $S_n = S_{M^n}$. Observe that the total number of possible strategies is finite and that

any non-optimal strategy occurs only finitely many time in the sequence $S_0, S_1, \ldots$. Thus, from some point on, every S_n is optimal. In reality, the desired n is typically not that large because the convergence of the computed costs towards the optimal costs is exponentially fast. For practical purposes, the iteration process halts when the additional gain is absorbed in rounding errors.

4 Experiments

We have conducted some experiments in Spec Explorer in order to evaluate which approach performs better on test graphs that arise from typical model programs, whether linear programming implemented by the simplex method or value iteration. Even for small examples the value iteration method has outperformed linear programming, and we are yet to find examples for which linear programming would be preferable. For some test graphs with several thousand vertices, the straightforward value iteration method as described above is also inadequate. We are currently investigating extensions of the value iteration method as well as the use of graph compression explained above. Table 1 compares the running times of the value iteration and simplex method for test graphs generated from the following two sample problems. Both examples are available in the Spec Explorer distribution [1].

Table 1. Comparison of value iteration and linear programming on sample problems.

Sample problem	Number of vertices in test graph	Value iteration running time	Simplex running time	Ratio simplex/ value iteration
Chat	40	4ms	33ms	8
Chat	92	25ms	370ms	15
Chat	225	100ms	7000ms	70
Bag	128	19ms	635ms	34
Bag	277	135ms	5600ms	41

Chat Described in Section 1.

Bag A multi-threaded implementation of a bag (multiset). Several concurrent users are allowed to add, delete and lookup elements from a shared bag. The example is a variation of a case study used in [23].

The different test graph sizes for Chat and Bag arise by varying the exploration settings using the Spec Explorer tool in a way that was illustrated on a smaller scale with that Chat model in Section 1. To motivate the relevance of the different cases in the table let us take a closer look at the two test graphs generated from the Bag model.

In the case with 128 vertices, the number of users (that equals the number of concurrent threads in the implementation) is 2 and the maximum allowed size of the bag is 1. This configuration yields an exploration of the state space with all the possible interleavings of the bag operations performed by the users. The test suite generated from the

test graph gives a global strategy to cover all the interleavings of the observable thread operations.

The case with 277 vertices corresponds to a situation when the maximum bag size is 2 elements and again there are 2 users. This provides a test suite that, in particular, covers the scenario when one of the users tries to remove all occurrences of a particular element and the other user tries to add that element to the bag.

The numbers 128 and 277 are in the range where the solutions are not obvious to the tester but the state graph is small enough for visual inspection and overview. The accepting state in both cases is the state where both users or threads are inactive.

The implementation of the bag example in the distribution [1] has a locking error that can only be discovered with at least two concurrent users accessing the bag.

5 Related work

Extension of the FSM-based testing theory to nondeterministic and probabilistic FSMs got some attention a while ago [14, 21, 28]. The use of games for testing is pioneered in [2]. A recent overview of using games in testing is given in [27].

An implementation that conforms to the given specification can be viewed as a refinement of the specification. In study [11], based on [3], the game view is proposed as a general framework for dealing with refining and composing systems. Models with controllable and observable actions correspond to interface automata in [11].

Model-based testing allows one to test a software system using a specification (a.k.a. model) of the system under test [5]. There are other model-based testing tools [4, 17–19, 24]. To the best of our knowledge, Spec Explorer is currently alone in supporting the game approach to testing. Our models are Abstract State Machines [15]. In Spec Explorer, the user writes models in AsmL [16] or in Spec# [6]. The theoretical foundations of the tool are described in [8].

In [20] several algorithms are described that generate optimal length-bounded strategies from test graphs, where optimality is measured by minimizing the total cost while maximizing the probability of reaching the goal, if the goal is reachable. The problem of generating a strategy with optimal *expected* cost is stated as an open problem in [20].

Solving negative and positive Markov decision problems with the total reward criterion are studied in [10] to address the basic problems of computing the minimum and maximum probability or the minimum and maximum expected time to reach a target set in a probabilistic system. In particular, the problem of computing the minimum expected time can be reduced to the negative Markov decision problem with the total reward criterion. In this paper we used Alfaro's Theorem 9 [10], which shows that linear programming works for negative MDPs after eliminating vertices from which the target state is not reachable, and that the optimal solution of the LP is unique.

One may wonder how transient stochastic games [12, Section 4.2] are related to transient test graphs. A transient stochastic game is a game between two players that will stop with probability 1 no matter which strategies are used. This condition gives rise to a proper subclass of transient test graphs where *all* strategies are reasonable. Recall that a test graph is transient if and only if there *exists* a reasonable strategy. An

unreasonable strategy is for example a strategy that takes you back and forth between two active states.

6 Conclusion

One of the main contributions of this paper is the identification of reachability games on transient test graphs as a fundamental notion in the area of testing of probabilistic systems. We show how known results, especially those on Markov decision process theory, can be used for a powerful effect in the context of testing. We provide some experimental results. And we worked out a careful self-contained exposition [7] of all the material. Finally let us note that this paper addresses a relatively easy case when all the states are known in advance. The more challenging (and important) case is on-the-fly or online testing where new states are discovered as you go [25]. In a sense, this paper is a warmup before tackling on-the-fly test strategy generation.

References

1. Spec Explorer. URL:http://research.microsoft.com/specexplorer, released January 2005.
2. 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.
3. R. Alur, T. A. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 163–178. Springer, 1998.
4. C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In Börger, Gargantini, and Riccobene, editors, *Abstract State Machines 2003*, volume 2589 of *LNCS*, pages 87–107. Springer, 2003.
5. M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a tool environment for model-based testing with AsmL. In Petrenko and Ulrich, editors, *Formal Approaches to Software Testing, FATES 2003*, volume 2931 of *LNCS*, pages 264–280. Springer, 2003.
6. M. Barnett, R. Leino, and W. Schulte. The Spec# programming system: An overview. In M. Huisman, editor, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
7. A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to test. Technical Report MSR-TR-2005-04, Microsoft Research, January 2005. Revised April 5, 2005.
8. C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Model-based testing of object-oriented reactive systems with Spec Explorer. Technical Report MSR-TR-2005-59, Microsoft Research, 2005.
9. K. Chatterjee, M. Jurdziński, and T. Henzinger. Simple stochastic parity games. In *CSL 03: Computer Science Logic*, Lecture Notes in Computer Science 2803, pages 100–113. Springer-Verlag, 2003.
10. L. de Alfaro. Computing minimum and maximum reachability times in probabilistic systems. In *International Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 66–81. Springer, 1999.

11. 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.
12. J. Filar and K. Vrieze. *Competitive Markov decision processes*. Springer-Verlag New York, Inc., 1996.
13. 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.
14. S. Gujiwara and G. V. Bochman. Testing non-deterministic state machines with fault-coverage. In J. Kroon, R. Heijunk, and E. Brinksma, editors, *Protocol Test Systems*, pages 363–372, 1992.
15. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
16. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 2005. To appear in special issue dedicated to FMCO 2003, preliminary version available as Microsoft Research Technical Report MSR-TR-2004-27.
17. A. Hartman and K. Nagin. Model driven testing - AGEDIS architecture interfaces and tools. In *1st European Conference on Model Driven Software Engineering*, pages 1–11, Nuremberg, Germany, December 2003.
18. 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.
19. V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Bourdonov. UniTesK: Model based testing in industrial practice. In *1st European Conference on Model Driven Software Engineering*, pages 55–63, Nuremberg, Germany, December 2003.
20. L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. In *ISSTA'04*, volume 29 of *Software Engineering Notes*, pages 55–64. ACM, July 2004.
21. A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Testing deterministic implementations from nondeterministic FSM specifications. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *IFIP TC6 9th International Workshop on Testing of Communicating Systems*, pages 125–140. Chapman & Hall, 1996.
22. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Mathematical Statistics. A Wiley-Interscience, New York, 1994.
23. S. Tasiran and S. Qadeer. Runtime refinement checking of concurrent data structures. *Electronic Notes in Theoretical Computer Science*, 113:163–179, January 2005. Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).
24. 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.
25. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *ESEC/FSE'05*, 2005.
26. J. von Neumann and O. Morgenstern. *The Theory of Games and Economic Behavior*. Princeton University Press, 1944.
27. M. Yannakakis. Testing, optimization, and games. In *Proceedings of the Nineteenth Annual IEEE Symposium on Logic In Computer Science, LICS 2004*, pages 78–88. IEEE, 2004.
28. W. Yi and K. G. Larsen. Testing probabilistic and nondeterministic processes. In *Testing and Verification XII*, pages 347–61. North Holland, 1992.