Optimal Strategies for Testing Nondeterministic Systems

Lev Nachmanson Margus Veanes Wolfram Schulte
Nikolai Tillmann Wolfgang Grieskamp
Microsoft Research, One Microsoft Way, Redmond, WA
{levnach,margus,schulte,nikolait,wrwg}@microsoft.com

ABSTRACT

This paper deals with testing of nondeterministic software systems. We assume that a model of the nondeterministic system is given by a directed graph with two kind of vertices: states and choice points. Choice points represent the nondeterministic behaviour of the implementation under test (IUT). Edges represent transitions. They have costs and probabilities. Test case generation in this setting amounts to generation of a game strategy. The two players are the testing tool (TT) and the IUT. The game explores the graph. The TT leads the IUT by selecting an edge at the state vertices. At the choice points the control goes to the IUT. A game strategy decides which edge should be taken by the TT in each state. This paper presents three novel algorithms 1) to determine an optimal strategy for the bounded reachability game, where optimality means maximizing the probability to reach any of the given final states from a given start state while at the same time minimizing the costs of traversal; 2) to determine a winning strategy for the bounded reachability game, which guarantees that given final vertices are reached, regardless how the IUT reacts; 3) to determine a fast converging edge covering strategy, which guarantees that the probability to cover all edges quickly converges to 1 if TT follows the strategy.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Reliability,Statistical methods; D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Algorithms,Reliability

Keywords

Probabilistic nondeterministic finite state machines, abstract state machines, optimal game strategies

1. INTRODUCTION

Modeling is a popular way of representing the behavior of a system. A common type of model in computing is a state graph, or finite state machine. State graphs are a useful way to think about software behavior and testing [6]. The application begins in some state (such as "main window displayed"), the user applies an input ("open file") and the software moves into a new state ("open file dialog displayed"). Model based software testing of an application can be viewed as traversing a path through the graph of the model and verifying that the implementation under test (IUT) works as predicted by the model.

Using state graphs or finite state machines (FSM) to test software systems is well understood and often used (see [22] for an excellent overview of testing theory using FSMs). However, most of the existing theory assumes that the FSM or state graph is deterministic, which means that the test harness has complete control of the IUT.

Modern systems, however, are often multithreaded if not distributed. For example the upcoming version of Windows will contain the Indigo system that provides a unified programming model and communications infrastructure for distributed applications [7]. Controlling and observing those systems is a very difficult task and often requires complex test architectures; see [2, 18, 21, 28] for recent model based testing tools architectures in this regard. In this paper we take a different approach: we simply model the implementation nondeterministically. And we don't assume that we can rely on controlling the nondeterminism (i.e. the scheduling) of the IUT. Instead we only observe the behavior of the implementation and generate test strategies that try to cover as much as possible of the behavior of the IUT. Since the state graph is nondeterministic the user guides the generated walks of the state graph by assigning costs and probabilities to the edges. This information is used during a walk of the graph, so that traversals are statistically more likely follow the higher probability edges. But since we also associate costs to edges we also want to minimize the total cost of traversals.

Markov chain probabilities (see for example [24, 12]), nondeterministic and probabilistic FSMs [1, 16, 29] and labelled transition systems [8, 27] have been proposed to test nondeterministic systems. But the important problem of how to find "optimal" traversals, i.e. those with minimal total costs and maximal probability, that lead from a start state to a set of goals states within a given number of steps has not been fully explored. The work in [30] is related to our work in this regard (see Section 7). However, that is exactly what testers usually want: Testers want to bring the system into a particular state within a length-bounded test sequence. And they want to know what it costs and which chance of success it has. Likewise, testers want to see as soon as possible (i.e. with the least costly test sequence) all possible behaviours distinguished by the model. But again, what is its likelihood of success (here all edges are covered) if nondeterminism is involved?

In this paper we propose three novel algorithms for testing nondeterministic systems. Test case generation in our setting amounts to generation of a game strategy. The two players are the testing tool (TT) and the IUT. The game explores some graph of states. The TT leads the IUT by choosing an edge in the graph everywhere except of the states where the application under test is nondeterministic. At the last states the control goes to the IUT. We suppose that the probabilities of the application choices are known and totally defined by the current graph vertex. To ensure short test cases we introduce a cost function on edges. A game strategy decides which edge should be taken by the TT in each state to win the game. This paper presents algorithms to do the following.

1. To determine an optimal strategy for the bounded reachability game, where optimality means maximizing the probability to reach any of the given goal nodes while at the same time minimizing the cost of traversal.

2. To determine a winning strategy for the bounded reachability game, which guarantees that goal nodes are reached, regardless how the IUT reacts.

3. To determine a fast converging edge covering strategy, which guarantees that the probability to cover all edges quickly converges to 1 if one follows the strategy.

All three algorithms are implemented in the AsmL Test tool [4], which is used by Microsoft product groups on a daily basis.

The remainder of this paper is structured as follows: In Section 2 we describe test graphs giving setting to the games. Algorithms for computing an optimal strategy for the bounded reachability game, for winning the bounded reachability game and for the fast converging edge coverage strategy are given in Section 3, 4, and 5, respectively. Section 6 presents aspects of the current implementation. Section 7 discusses related work.

2. TEST GRAPHS

We use test graphs to describe nondeterministic systems. A *test graph* G has a set V of *vertices* and a set E of directed *edges*. There are two functions *source* and *target* from E to V providing for each edge the *source* and the *target* of that edge, respectively. V is divided into two disjoint sets of vertices: *states* (*States*) and *choice points* (*CP*) . There is a *probability function* p mapping edges exiting from choice points to non-negative real numbers such that, for every choice point v,

$$\sum_{e \in E, source(e)=v} p(e) = 1. \qquad (1)$$

Notice that this implies that for every choice point there is at least one edge starting from it. Finally, there is a *cost function cost* from edges to non-negative reals. Formally we denote G by the tuple

$$(V, E, CP, source, target, p, cost).$$

We sometimes use the shorthand notation (u, v) for an edge with source u and target v if the identity of the edge is irrelevant. Notice that in general there may be several edges with the same source and target but with different costs and probabilities. This corresponds to several actions being enabled from a given source state that nevertheless lead to the same target state.

3. BOUNDED REACHABILITY GAME

Given a test graph G, a *bounded reachability game* over G is given by a start vertex s, a non-negative maximum number n of moves and a set of *goal vertices* $P \subset States$. We denote a bounded reachability game by the tuple (s, n), excluding P from the notation since P is always fixed.

There are two players in the game, a tester (TT) and an implementation under test (IUT). The current state of the game is given by a vertex v and a number k of moves made so far. Initially $v = s$ and $k = 0$. The players make moves according to the following rules.

```
if (v is a choice point){
   if (k ≥ n) {TT loses and the game stops;}
   else {
     IUT chooses an edge (v,u);
     v=u; k=k+1;
   }
}
else if (v in P) {TT wins and the game stops;}
else if (k ≥ n)  {TT loses and the game stops;}
else {
  TT chooses an edge (v,u) or loses if no choice is possible;
  v=u; k=k+1;
}
```

It is assumed that IUT chooses an edge e with probability $p(e)$. Every particular run of a game over G produces a path in G which we will call a *game path*. A game produces no paths if TT wins or loses at the start of the game. Without loss of generality for the bounded reachability game we may assume that there are no edges exiting from P, since the game always terminates at this point.

3.1 Strategies for bounded reachability game

A *strategy* for a bounded reachability game (v, n) over a test graph

$$G = (V, E, CP, source, target, p, cost)$$

is a function S from $States \times \{0, \ldots, n\}$ to $E \cup \{null\}$, where *null* is a special value indicating that the strategy is not defined for the given arguments and TT loses at this point. For any state v and $k \leq n$, either $S(v, k) = null$ or $S(v, k)$ is an edge exiting from v.

A strategy S is used by TT to choose an edge $S(v, n-k)$ at the k'th step of the game. In other words, $S(v, m)$ is the edge to be used when at most m moves remain to the end of the game. If TT follows a strategy S then the body of the last else case above becomes the following rule:

```
let e = S(v,n-k);
if (e = null) {TT loses;}
else {v=target(e);k=k+1;}
```

We define a bounded reachability game (v, n, S) to be the game (v, n) where TT follows the strategy S.

3.2 Comparing strategies

We would like to measure bounded reachability game strategies and compare them. To this end, we define two functions on strategies: *Prob* and *Cost*. Consider a bounded reachability game (v, n, S). Recall that TT wins if a goal vertex is reached in at most n steps, and loses in the opposite case.

Let $Prob_S(v, n)$ denote the probability of TT winning the game (v, n, S). One can see that

$$Prob_S(v,n) = \begin{cases} 1, \text{if } v \in P; \\ 0, \text{if } v \notin P \text{ and } (n = 0 \text{ or } S(v,n) = \mathit{null}); \\ Prob_S(u, n-1), \text{if } v \notin P, \\ \quad v \text{ is a state, } n > 0 \text{ and } S(v,n) = (v,u); \\ \sum_{(v,u)\in E} p((v,u)) Prob_S(u, n-1), \\ \quad \text{if } v \text{ is a choice point and } n > 0. \end{cases} \tag{2}$$

The last equation holds since an event of playing a game with n steps starting from a choice point can be considered as a disjoint sum of events of choosing an edge starting from the choice point and then playing the $n-1$ step game starting from the target of the edge.

The cost of a game path is the sum of the costs of all its edges. The cost of a game (v, n, S), $Cost_S(v, n)$, is the cost of a game path with the highest cost, or 0 if there are no game paths. One can formally describe the cost function as follows.

$$Cost_S(v,n) = \begin{cases} 0, \text{ if } v \in P \text{ or } S(v,n) = \mathit{null} \text{ or } n = 0; \\ cost((v,u)) + Cost_S(u, n-1), \\ \quad \text{if } v \text{ is a state not in } P, \\ \quad S(v,n) = (v,u) \text{ and } n > 0; \\ \max_{(v,u)\in E} cost((v,u)) + Cost_S(u, n-1), \\ \quad \text{if } v \text{ is a choice point and } n > 0. \end{cases} \tag{3}$$

As a single measure of a strategy S for fixed n and P we consider the pair of functions $(Prob_S, Cost_S)$ where the domain of each function is $V \times \{0, \ldots, n\}$. Let $[0, 1]$ denote the segment of non-negative real numbers at most 1, and let $\Re$ be the set of non-negative real numbers. By a *performance* value we mean an element in the following set of pairs of real numbers.

$$Pairs = [0,1] \times \Re.$$

Consider a total preorder (transitive and reflexive relation) $\preceq$ ("at least as good as") on performance values. Let the game bound n and the set P of goal states be fixed. We say that a strategy S *improves* a strategy S' (with respect to $\preceq$), denoted by $S \preceq S'$, if for every vertex v and $k \le n$,

$$(Prob_S(v,k), Cost_S(v,k)) \preceq (Prob_{S'}(v,k), Cost_{S'}(v,k)).$$

In other words, S improves S' if the performance of S from every vertex and bound is at least as good as that of S'. Clearly improvement is a preorder. We say that two strategies are *equally good* if they improve each other. A strategy S (for a given game) is *optimal* (for $\preceq$) if every other strategy that improves it is equally good as S. Notice that there are only finitely many strategies for a given game, several of which may be optimal. Notice also that not all optimal strategies are necessarily equally good because the improvement preorder may be partial. In other words, one strategy may perform better from one vertex whereas another strategy may perform better from another vertex.

Example 1. To motivate this measure on strategies let us consider a preorder $\preceq$ on *Pairs* such that $(0, c)$ is a greatest element for any $c \ge 0$ (TT cannot win) and that if $p' > 0$ then $(p, c) \prec (p', c')$ if either $p > p'$ (the probability is higher) or, $p' = p$ and $c < c'$ (the probability is the same but the cost is lower). In this case an optimal strategy gives for TT the maximum probability to win (if winning is possible at all) while minimizing the maximal path cost. If TT cannot win then all strategies have winning probability 0 and all strategies are equally good and optimal.

Another possibility for measuring strategies would be combining of the cost and the probability into a single expected cost value. However, the best expectation does not necessarily guarantee that in a particular run of the game the path cost is low. For example if the cost of edges corresponds to the time of the transition execution then our approach generates test strategies running not longer then some known time interval. In this sense the maximal cost is a safer criterion than the expected value.

Definition 1. A total preorder $\preceq$ on *Pairs* is *acceptable* if the following conditions hold.

1. For all $c \ge 0$ the pair $(0, c)$ is a greatest element in *Pairs*, i.e. for all $x \in Pairs$, $x \preceq (0, c)$.
2. Let $(P_i, C_i), (P'_i, C'_i) \in Pairs$, $p_i \in [0,1]$, $c_i \in \Re$, for $1 \le i \le m$, such that $\sum_i p_i P_i \le 1$ and $\sum_i p_i P'_i \le 1$. If
$$(P_i,\ C_i) \preceq (P'_i,\ C'_i), \quad \text{for } 1 \le i \le m,$$
then
$$(\sum_i p_i P_i,\ \max_i(c_i + C_i)) \preceq (\sum_i p_i P_i^{'},\ \max_i(c_i + C_i^{'})).$$

The preorder defined in Example 1 is acceptable. It follows from the fact that (0,c) is a greatest element for any $c \ge 0$ and from the observation that if $a \le a'$ and $b \le b'$, then $a + b \le a' + b'$ and $\max(a, b) \le \max(a', b')$.

3.3 Reachability Strategy Calculation

We provide an algorithm for computing strategies. We start by providing the intuition of the algorithm. We then explain the algorithm in detail using an intuitive pseudo code notation. The input to the algorithm is a bounded reachability game and it computes a strategy `S`, a probability function `Pr` and a cost function `C`.

Proposition 1 establishes the correctness of the algorithm, i.e. that the computed probability and cost functions coincide with $Prob_S$ and $Cost_S$, respectively.

The algorithm builds an optimal strategy for acceptable preorders. This is proved in Theorem 1.

The intuition of the algorithm is as follows. The algorithm performs a breadth-first backward search with some repetitions. It starts from the set of goal states `P` that is the initial value of its frontier and iterates a main calculation phase n times, where n is the bound of the game. During the k'th iteration it updates the frontier and calculates the next value of the strategy, probability function and cost function. Each iteration preserves the optimality of the strategy computed so far if the preorder is acceptable.

The algorithm has the following variables. The variable `front` is the frontier of vertices that is initially equal to P. The variable `newfront` is the new frontier that is produced by the current step of the algorithm; `newfront` is initially empty. The set `newfront` is also used to avoid unnecessary recalculations of `Pr` and `C` for the choice points.

The procedure `Initialize` establishes the initial state of the algorithm.

```
Initialize () {
   front=P; newfront=∅;
   foreach (v ∈ P)
         Pr(v,0)=1;

   foreach (v ∈ V - P)
         Pr(v,0)=0;

Definition   foreach (v ∈ V)
         C(v,0)=0;

   foreach (v ∈ States)
         S(v,0) = null;
}
```

The main body of the algorithm is a bounded loop of n iterations. During the i'th iteration of the loop, edges ending in the current frontier are traversed backwards and values of $S(v,i)$, $\mathtt{Pr}(v,i)$ and $C(v,i)$ are updated.

```
   for (i=1; i≤n; i=i+1){
       PropagateChanges(i);

       foreach ( edge in E where target(edge) ∈ front )
         TraverseEdge(edge,i);

       front=newfront;
       newfront=∅;

   }
```

The procedure `PropagateChanges` prepares the i'th step of the main loop, either using the result of the previous step, if $i > 1$, or if $i = 1$, using the initial settings. The argument i of `PropagateChanges` must be in $\{1, \ldots, n\}$.

```
PropagateChanges(i){
    foreach(v ∈ V){
       Pr(v,i)=Pr(v,i-1); C(v,i)=C(v,i-1)
    }
     foreach(v ∈ States)
        S(v,i)=S(v,i-1)
 }
```

The procedure `TraverseEdge` takes two parameters; an edge `e` and the current step number i . Let u be the source of `e`. The procedure may update $\mathtt{Pr}(u,i)$ and $C(u,i)$ for some vertex u, and it may also update $S(u,i)$ for some state u.

Assume u is a choice point. If $u \notin$ `newfront` then $\mathtt{Pr}(u,i)$ and $C(u,i)$ are updated based on the values of $\mathtt{Pr}(w,i-1)$ and $C(w,i-1)$ for all edges $(u,w) \in E$. If $u \in$ `newfront` then the procedure exits without making any changes, otherwise u is inserted into `newfront` .

Assume u is a state. If including the edge e in the strategy improves the performance from u, then the old value of $S(u,i)$ is replaced by e. The probability and the cost functions are updated accordingly and u is inserted into `newfront`.

```
TraverseEdge(e,i){
   let u = source(e)
   let v = target(e)
   if (u ∈ CP){
      if (u ∈ newfront) return;

L1:   Pr(u,i) = sum{prob(d)Pr(target(d),i-1) |
                                d ∈ E where source(d)=u};
L2:   C(u,i) = max{cost(d) + C(target(d),i-1) |
                                d ∈ E where source(d)=u};
      insert u into newfront;
   }
   else if(Improving(e,i)){
     S(u,i)  = e;
     Pr(u,i) = Pr(v,i-1);
L3:  C(u,i)  = cost(e)+C(v,i-1);
     insert u into newfront;
   }
}
```

The function `Improving` checks if it is beneficial to set $S(u,i)$ to `e` by comparing the new performance value with the old one. Notice that the initial performance values are set to $(0,0)$ that is a greatest element in $\prec$, given that $\prec$ is acceptable.

```
bool Improving(e, i){
   let (u,v)=e;
   return
   (Pr(v,i-1),cost(e)+C(v,i-1))≺(Pr(u,i),C(u,i));
}
```

3.3.1 Correctness

The following proposition establishes the correctness of the algorithm regarding the computation of the probability and cost functions.

Proposition 1. *If the preorder is acceptable then the algorithm builds a strategy S such that*

$$Prob_S = Pr$$

and

$$Cost_S = C.$$

Proof. By induction on n. The base case follows from the definitions.

Suppose that the proposition is true for $n-1$. Let us prove that $Prob_S(v,n) = Pr(v,n)$ for any $v \in V$. Let u be a state and $(u,v) = S(u,n)$. The probability of TT to win in the game (u,n,S) is $Prob_S(u,n) = Prob_S(v,n-1)$ because TT makes a move $S(u,n)$ and then the game $(v,n-1,S)$ is played. By induction hypothesis $Prob(S)(v,n-1) = Pr(v,n-1))$. The procedure `TraverseEdge` sets $Pr(u,n)$ to $Pr(v,n-1)$. It is proven that $Prob_S(u,n) = Pr(u,n)$ for $u \in States$.

Let u be a choice point then

$$\begin{aligned} Prob_S(u,n) &= \sum_{(u,v)\in E} p((u,v))Prob_S(v,n-1) \\ &= \sum_{(u,v)\in E} p((u,v))Pr(v,n-1) = Pr(u,n). \end{aligned}$$

The first equality holds by equation (2), the second one by the induction hypothesis and the third one by the construction in the procedure `TraverseEdge`. We have proven that $Prob_S = Pr$.

Let us show that $Cost_S = C$. Recall that $Cost_S(u,i)$ is the cost of the most expensive game path of the game

(v, i, S). The lines L2 and L3 in `TraverseEdge` ensure that if $Costs_S(v, n-1) = C(v, n-1)$ for any $v \in V$ then $Costs_S(u, n) = C(u, n)$ for any $u \in V$. $\square$

It is worth noting that the proof of Proposition 1 uses only Case 1 of Definition 1.

3.3.2 Optimality

During the i'th iteration of the algorithm, for a given state u, $S(u, i)$ is set in such a way that

$$(Prob_S(u, i), Costs_S(u, i)) = \min_{(u,v)\in E}(Prob_S(v, i-1),\ cost((u, v)) + Costs_S(v, i-1)),$$

where the minimum is taken with respect to the total preorder $\preceq$ on *Pairs*.

Notice that the values of $Pr(u, i)$ and $C(u, i)$ cannot be influenced for a choice point u because they are determined by (2) and (3). This implies that we cannot guarantee optimality of the computed strategy if the preorder is not acceptable. In particular Case 2 of Definition 1 is used in the proof of Theorem 1.

Theorem 1. *If $\preceq$ is an acceptable preorder then the strategy S computed by the algorithm is optimal for $\preceq$.*

Proof. By induction on the number of game steps. The constructed strategy S is minimal for any game with 0 steps because the probability and the cost functions do not depend on the strategy. This proves the base case of the induction hypothesis.

Suppose that the theorem is true for games with $n-1$ steps. Let R be any strategy. We need to show that

$$(Prob_S(u, n), Costs_S(u, n)) \preceq (Prob_R(u, n), Cost_R(u, n)). \tag{4}$$

There are two cases.

1. Assume u is a state. If $R(u, n) = null$ and $u \notin P$ then

$$(Prob_R(u, n), Cost_R(u, n)) = (0, 0)$$

and by Definition 1.1, we have

$$(Prob_S(u, n), Costs_S(u, n)) \preceq (0, 0).$$

If $R(u, n) = null$ and $u \in P$ then

$$(Prob_R(u, n), Cost_R(u, n)) = (1, 0) = (Prob_S(u, n), Costs_S(u, n)).$$

And thus (4) follows by reflexivity of $\preceq$.

Now suppose that $R(u, n) = (u, w)$ for some $w \in V$. Then we have

$$\begin{aligned}
&(Prob_S(u, n), Costs_S(u, n)) \\
&= \quad (5)\\
&\min_{(u,v)\in E}(Prob_S(v, n-1), cost((u, v)) + Costs_S(v, n-1)) \\
&\preceq \quad (6)\\
&\min_{(u,v)\in E}(Prob_R(v, n-1), cost((u, v)) + Cost_R(v, n-1)) \\
&\preceq \quad (7)\\
&(Prob_R(w, n-1), cost((u, w)) + Cost_R(w, n-1)) \\
&= \quad (8)\\
&(Prob_R(u, n), Cost_R(u, n)).
\end{aligned}$$

Equality (5) holds by construction of S. Equation (6) holds by the induction hypothesis. Equation (7) is true because $\preceq$ is total and a least element (with respect to $\preceq$) of a set is not greater than any of its elements. Equality (8) follows from the equations (2,3). Thus (4) follows by transitivity of $\preceq$.

2. Assume u is a choice point. Consider all edges $e_i = (u, v_i)$ exiting form u, where $1 \le i \le m$. Let

$$\begin{aligned}
p_i &= p(e_i),\\
c_i &= cost(e_i),\\
P_i &= Prob_S(v_i, n-1),\\
C_i &= Costs_S(v_i, n-1),\\
P'_i &= Prob_R(v_i, n-1),\\
C'_i &= Cost_R(v_i, n-1).
\end{aligned}$$

Then

$$\begin{aligned}
&(Prob_S(u, n), Costs_S(u, n))\\
&= (\sum_i p_i P_i, \max_i c_i + C_i) \quad (9)\\
&\preceq (\sum_i p_i P'_i, \max_i c_i + C'_i) \quad (10)\\
&= (Prob_R(u, n), Cost_R(u, n)) \quad (11)
\end{aligned}$$

Equations (9) and (11) follow from (2) and (3), equation (10) follows from the induction hypothesis and Definition 1.2.

The theorem follows by the induction principle. $\square$

If k is the in-degree of the test graph (i.e. for any vertex v the number of edges entering into v is at most k) then the running time of the algorithm is $O(n|V|k)$ steps. Since $k \le |E|$ the complexity of the algorithm is $O(n|V||E|)$.

3.3.3 Application to Blackjack

Note that if a bounded reachability game graph is acyclic then for some step N of the algorithm $front$ becomes empty and for any $m > N$ and state v

$$S(v, N) = S(v, m).$$

In other words, there is a bound after which the strategy cannot be further improved. This situation arises for the card game Blackjack. For completeness we give the basic rules of Blackjack in the appendix. We used the algorithm described above to obtain an optimal strategy for a player playing Blackjack against a dealer with one deck of cards. We simplified the game rules by omitting the player options "Split" , "Double down" and "Insurance" which are not explained here.

The states of the underlying graph are given by Blackjack situations where the player can decide between the actions "Hit" or "Stand", and the Blackjack situations where the game stops and the player either wins or loses. The states where the game stops do not have any outgoing edges. The goal states are the states where the player wins. The choice points are the Blackjack situations where the dealer draws a card. The probability function p on edges is naturally given by the probability of a specific card being drawn from the current deck. The cost function *cost* on edges is 1 for all edges. So the cost of the game is the maximum total number of moves that the player and the dealer make. For the preorder $\preceq$ we used the lexicographic order on pairs as given in Example 1.

Since the number of steps in Blackjack is not very relevant the player should follow the strategy $S(v, N)$, for any state v, saying "Hit" or "Stand" and N being the number of cards. The function $Prob_S$ provides interesting information for the player on the chances of winning from a given state of the game; for example with the dealer hand of Eight, and the player hand of Eight and Nine the strategy advises to "hit" and gives the probability 0.2354316 of winning.

4. WINNING STRATEGIES

In some cases it is possible to reach a goal state with full certainty, even though one has to deal with choice points on the way. The simplest example of this is a diamond-like behavior pattern where there are two edges exiting from a choice point but from the resulting target states all edges lead to a common goal state. This naturally comes up as a subproblem in testing nondeterministic systems: for example in generating a test case to reach a state satisfying a given property, or in generating a state identifying transfer tree for fault-detection of nondeterministic FSMs [30].

In terms of games we have the following situation. Let the test graph and the set of goal states be fixed. From some states v in the graph and for some bound $n \geq 0$ TT can win the game (v, n) with probability 1, let us call such states *winnable*. We describe an algorithm that finds all winnable states and provides a minimal cost winning strategy from those states. The algorithm is a variation of Dijkstra's shortest path algorithm and reduces to it in the special case when the graph contains no choice points.

4.1 Optimal Winning Strategy Algorithm

The input to the algorithm is a test graph

$$G = (V, E, CP, source, target, p, cost)$$

and a set of goal states P. Without loss of generality, we assume that there are no edges from choice points whose probability is 0. If such edges were present they could simply be removed, since they would never be present in any game path.

The output of the algorithm is a strategy S that provides a minimal cost strategy from all winnable states. The algorithm uses the following variables. The variable d is a *cost function* defined on V that initially maps all goal states to 0 and all other vertices to the special value ∞ ("infinity"). The variable R is a set of all *resolved* vertices, that is initially empty. The variable Q is a *priority queue* of vertices where the priority of a vertex v is the value of $d(v)$ (the lower the value of $d(v)$ the higher the priority). Vertices are removed from the queue in priority order and inserted into R, with vertices with higher priority being removed first. Initially, Q contains all the states. The set of successors to a choice point u is denoted by $succ(u)$,

$$succ(u) = \{target(e) : source(e) = u, e \in E\}$$

A choice point u is added to Q and $d(u)$ is updated only when all of the successors of u have been resolved (belong to R). Note that u itself may be included in $succ(u)$, in which case u would never be resolved. If a choice point is never resolved it means that there is no winning strategy to P going through u.

The variable S, mapping *States* to $E \cup \{Null\}$, is the computed strategy. Initially S maps all vertices to *null*. In this algorithm the strategy calculation does not depend on the number of moves. The procedure `Initialize` establishes the described initial values for the variables of the algorithm.

```
Initialize(){
   R = ∅;
   Q = States;
   foreach (v ∈ P) d(v)=0;
   foreach (v ∈ V-P) d(v)=∞;
   foreach (v ∈ V) S(v)= null;
}
```

The main body of the algorithm `CalculateWinningStrategy` removes vertices from the queue in priority order and calls the procedure `Relax` on edges entering the selected vertex until the queue becomes empty. This loop is identical to the main loop of Dijkstra's algorithm (see e.g. [10, Section 25.2]), except that the direction of the edges is reversed and that there are multiple goal states. (One can assume, without loss of generality, that there is a single goal state.)

```
CalculateWinningStrategy(){
  while (Q is not empty){
    v = Q.pop();
    R = R ∪ {v};
    foreach (e ∈ E where target(e)=v)
      Relax(e);
  }
}
```

The procedure of "relaxing" and edge $e = (u, v)$ is similar to the definition used in [10].

If u is a state and the value of the cost estimate $d(u)$ from u to P decreases by passing through e, then the strategy from u is updated to use the edge e.

The new part is the special treatment of choice points. If u is a choice point then u is added to the queue only if all of its successors have been resolved. The shortest distance estimate from u is updated according to the worst case scenario, i.e. considering all the possible edges that the opponent can choose from u and assigning to $d(u)$ the maximum cost resulting from those choices.

```
Relax((u,v)){
    if (u ∈ States)
      if (d(u) > d(v) + cost((u,v))){
        d(u) = d(v) + cost((u,v));
        S(u) = (u,v);
      }
    else
      if (succ(u) ⊆ R){
        d(u)= max{cost((u,w))+d(w) | (u,w) ∈ E};
        Q = Q ∪ {u};
      }
}
```

4.2 Analysis

We call a strategy *certain* if for some number n for every winnable vertex the strategy leads to P in no more than n steps and the strategy is *null* on non-winnable vertices. Let S be a certain strategy. For any vertex v we define $C_S(v)$ as the maximum of the game path costs in the game starting from v following S if v is a winnable vertex and ∞ otherwise. A strategy S is *optimal* if it is certain, and for any other certain strategy S', $C_S(v) \leq C_{S'}(v)$ for any vertex v.

THEOREM 2. *Let S and d be computed by the algorithm above. Then S is optimal and $d = C_S$.*

PROOF. The proof is similar to the proof of the correctness of Dijkstra's algorithm. First, we note the following fact about choice points: when a choice point u is added to Q its d-value is optimal. To see that, consider the first time when a choice point u is added to Q. This means that all of its successors, say $v_1, \ldots, v_n$, are states in R, and by the proof of Dijkstra's algorithm, the values of $d(v_i)$ are the shortest distances to P. Now u is added to Q and $d(u)$ is assigned the maximum of $cost((u, v_i)) + d(v_i)$, that is optimal and can not be improved. Thus, we may consider u as a state vertex whose only outgoing link is (u, v_j) (corresponding to a move made by the opponent that yields the cost $d(u)$). We may now repeat the argument to any subsequent additions of choice points to Q.

The key invariant that is preserved by the algorithm is that every time a vertex v is removed from Q and inserted into R, the strategy leading from that vertex to P is optimal and $d(v)$ is the corresponding cost. By the argument above, one can reduce the proof to the standard proof of Dijkstra's algorithm. $\square$

One can also show, by a careful selection of the data structures, that the complexity of the algorithm is the same as the complexity of Dijkstra's shortest path algorithm.

4.3 Remarks

The algorithm presented above accomplishes really two tasks: 1) it determines what states are winnable, and 2) it calculates an optimal strategy for the winnable states. Regarding (2) the algorithm is a straightforward extension of Dijkstra's shortest path algorithm, by making similar use of a priority queue. Regarding (1), with no choice points the problem is simply the graph accessibility problem, with choice points the problem is equivalent to the alternating graph accessibility problem AGAP [19] that is also essentially the same as the two-person game problem first addressed in [20]. To make the connection clear we show how the problem is related to AGAP following the formulation in [14]. Input to AGAP is a directed graph $D = (V, A)$ and two vertices s and t, in D each vertex is marked either as an *and*-vertex or and *or*-vertex. The problem is to decide whether t is reachable from s. A *predecessor* of a vertex v in D is a vertex u such that $(u, v) \in A$. Vertex t is *reachable* from s in D if a "pebble" can be placed on t using the following rules.

- One can place a pebble on s and on any vertex with no predecessors.
- A pebble can be placed on an and-vertex if it can be placed on all of its predecessors.
- A pebble can be placed on an or-vertex if it can be placed on at least one of its predecessors

Given a test graph G and a set of goal states P, construct D so that D contains all the vertices of G. The choice points of G are and-vertices in D and the states of G are or-vertices in D. For each edge (u, v) in G, D has the edge (v, u), i.e. the direction of the edges is simply reversed. For each or-vertex v in D that has no predecessors add the loop (v, v) to avoid initial pebbles. Finally, add a new or-vertex s to D, and for each goal state v in P, add an edge (s, v) to D. On can easily see that a state t is winnable if and only if t is reachable from s in D. Notice that the placement of pebbles in D is basically the same as backward search in G starting from the goal states.

5. EDGE COVERAGE GAME

The classical transition tour method or T-method is perhaps the most widely used method in practical model-based testing applications and is supported by virtually all model-based testing tools, see e.g. [25]. One of the reasons for its popularity is that it provides coverage of the transitions of the model that is an easy-to-understand metric for testers and at the same time provides a cost-optimal test suite.

We describe here a heuristic to generate a test suite in form of a game strategy so that the total cost of the test suite may be optimized and the generated strategy (potentially) covers the transitions of the model. In the case when the model is deterministic the approach reduces to the T-method. This approach is used in one of the key test case generation utilities of the AsmL tester tool [4]. We make here the usual assumption that the system under test has a Reset method implying that the underlying graph is strongly connected, (i.e. by resetting you can always return to the initial state).

Suppose we have a strongly connected test graph G as defined in 2. *A coverage game* between TT and IUT is the game where TT chooses an edge at the state vertices and IUT at the choice point vertices and TT tries to cover as many edges as possible. We suggest a heuristic which uses a tour through the graph, for example Chinese Postman tour, to produce a strategy for TT. Recall that a tour of G is a cycle containing all edges of G and a Chinese Postman tour is a minimal cost tour. A Chinese Postman tour can be computed efficiently, see for example [13]. Suppose we have such a tour and we cut it into sequences at choice points. Now the strategy of TT can be as the follows. If v is a state vertex then choose a sequence containing it and follow the sequence until the end which is a choice point. At the choice point IUT chooses an edge randomly and if the edge target is a state then TT proceeds with any sequence continuing that edge. Let us explain the heuristic more formally and estimate the probability of TT covering all edges by following this strategy.

Let T be a sequence of edges composing a tour covering G. We can assume without loss of generality that there is at least one choice point and that the start and the end vertex of the tour is a choice point. The sequence T can be represented as a concatenation of sequences $T_i, 1 \leq i \leq n$, such that the source of the first edge of T_i is a choice point, the target of the last edge of T_i is a choice point, and there are no choice points internal to T_i. Let TS be the set of all T_i. The strategy of the coverage game is given by the pseudo code of `Cover`. Normally the actual start vertex is some initial state and not a choice point, in which case TT first brings the selected vertex to a choice point by following some T_j containing the initial state.

`Cover` has two parameters; the selected vertex v which is a choice point, and an integer k containing the number of loop iterations.

```
Cover(vertex v,integer k){
   while (k > 0){
       k=k-1
```

```
        let e=edge chosen by IUT
        let es=choose randomly any sequence
               from TS starting with e;
        cover every edge of es;
        v=end of es;
    }
}
```

For each choice point v and every $i \in \{1..n\}$ let us denote by $p_{i,v}$ the probability that the sequence T_i is be covered by `Cover(v,n)`. Since T is a cycle $p_{i,v} > 0$ for every $i \in \{1..n\}$ and for every choice point v. Let

$$p = \min_{i \in \{1..n\}, v \in CP} p_{i,v}.$$

Obviously, $p > 0$.

PROPOSITION 2. *Let m be a natural number. For any choice point v the procedure `Cover(v,mn)` covers all edges of G with probability at least $1 - n(1-p)^m$ and therefore the probability to cover all edges converges to 1 when m goes to infinity.*

PROOF. It is enough to prove that `Cover(v,mn)` covers all sequences T_i with probability at least $1 - n(1-p)^m$. Let $v_1 = v$, $v_2 = v$ after n steps of `Cover` and so on and v_m is the value of v after $(m-1)n$ steps of `Cover`. The probability that the sequence T_k is not covered by mn steps of `Cover` for some $k \in \{1..n\}$ is

$$\prod_{i \in \{1..m\}} (1 - p_{k,v_i}) \leq (1-p)^m.$$

The probability that at least one sequence T_k is not covered by mn steps of`Cover` is not more than $n(1-p)^m$. Therefore the probability that all sequences T_k are covered by nm steps of `Cover` is not less than $1 - n(1-p)^m$. □

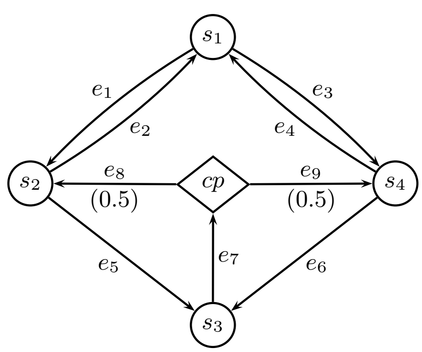

Figure 1: A sample test graph; all vertices are states except of the one labelled *cp* that is a choice point. The probability function is defined on the edges e_8 and e_9 and maps each of the edges to 0.5. The *cost* of each edge is 1

To illustrate the technique described here let us consider the coverage game on the graph in Figure 1. The edge sequence

$$[e_8, e_2, e_1, e_5, e_7, e_9, e_4, e_3, e_6, e_7]$$

is a Chinese Postman tour of the graph that starts and ends in *cp*. The sequence is divided into following subsequences:

$$[e_8, e_2, e_1, e_5, e_7] \quad [e_9, e_4, e_3, e_6, e_7]$$

Suppose that the initial state is s_1. To start with, TT can follow either $[e_1, e_5, e_7]$ or $[e_3, e_6, e_7]$ to bring the selected vertex to *cp*. Consequently, TT will follow the first sequence if IUT takes the edge e_8 or the second sequence if IUT takes the the edge e_9.

6. IMPLEMENTATION

The algorithms presented in the paper are used internally by the test case generation and the conformance checking utilities of the AsmL tester tool [4]. The tool is available from the AsmL web site [3].

In the AsmL tester tool users model nondeterministic systems using Abstract State Machines (ASMs) [17]. Similar to ordinary programs, ASMs can have potentially infinite space. The tool generates an FSM from the model exploring, generally speaking, only a part of its state using equivalence classes on states, that are arbitrary user defined relations in terms of model variables, and other techniques [15].

In earlier versions of the tool we use the traditional approach of generating test sequences. For example for transition coverage we build a Chinese Postman tour. The implementation of the Chinese Postman algorithm in the tool follows the scheme from [13]; the graph is extended to a balanced graph by adding augmenting paths from the vertices with shortage of outgoing edges (D-) to vertices with shortage of incoming edges (D+) and then an Euler tour is built on the extended graph thus producing a Chinese Postman tour on the original graph. The chosen augmenting paths can be viewed as a minimal weighted matching of the bipartite graph of all paths between D- and D+. To solve the minimal weighted matching problem the tool utilizes [23]. While a technique of following a test sequence works great for testing deterministic systems, it needs extensions to work for nondeterministic systems.

This leads us to rephrase the testing of nondeterministic systems as games. Now the AsmL test tool generates a test suite in form of a bounded reachability game strategy that tries to cover the edges using the approach described in Section 5 augmented with winning strategies to reach specific states generated by the algorithm described in Section 4. The conformance checking utility of the tool is then used to perform runtime verification [5] by executing the test suite against an IUT that is a .NET assembly. The user may configure several heuristics for controlling how many times the choice points should be revisited in case some edges are not taken by the IUT. After each step (edge taken by either the tester or the IUT) additional conformance relations (that are arbitrary user defined Boolean relations in terms of model variables and implementation variables) are used to compare the states of the model and the IUT at binary level. If a discrepancy is encountered the test execution fails and an error report is produced.

7. RELATED WORK

The bulk of the testing literature deals with deterministic FSMs [22, 26] where test cases are considered to be action sequences and the tester is assumed to be in full control of the state of the IUT. Since many of our applications are nondeterministic we needed to take a different approach in our testing framework. It seemed natural for us to look at test cases as game strategies.

Extension of the FSM based testing theory to nondeterministic and probabilistic FSMs [1, 16, 29] in particular for testing protocols got some attention ten years ago. However with the advent of multithreaded and distributed systems, it recently got more attention. In this regard, the recent paper [30] is closely related to our work. The approach taken in [30] does not use games though, instead, the notion of a transfer sequence is extended to the notion of a *transfer tree*, which is similar to the way test cases are considered in the labelled transition system based testing theory [8, 27]. The notion of a transfer tree corresponds to our notion of a game strategy. The two main problems addressed in [30] that are related to the first two problems that we address in this paper are MAW: find a transfer tree with minimum average weight, and MWH: find a transfer tree with minimum weighted height. The bounded reachability strategy calculation is quite different from MAW, we do not consider the average cost but rather optimality with respect to a given preorder on performance values. MWH on the other hand is essentially the problem of finding an optimal winning strategy. The algorithm we propose was developed independently and is different from [30, Minimum-weighted-height algorithm] in that our algorithm is a direct extension of Dijkstra's shortest path algorithm. We also show a close connection of the problem itself with the alternating graph accessibility problem [20] (see Section 4.3). To the best of our knowledge, the use of the transition tour method as presented in Section 5 for nondeterministic state machines has not been addressed in the literature.

If there is no upper bound imposed on the number of steps, then one can use another method for computing strategies for the reachability game based on [9]. While the method presented in [9] can be used to calculate a strategy that maximizes the probability of winning, here we are also interested in taking the cost into account. Consider a test graph $G = (V, E, CP, \mathit{source}, \mathit{target}, p, \mathit{cost})$, as defined in Section 2, where $V = \mathit{States} \cup CP$ and $\mathit{States} \cap CP = \emptyset$, and consider a set of *goal vertices* $P \subseteq \mathit{States}$. A strategy in this context is a function from States to $E \cup \{\mathit{null}\}$. Let us briefly describe the method under the assumption that P is reachable from every vertex. (This assumption is satisfied in a common situation of being able to reset the system in any state and transition back to the initial state.) Consider a fixed strategy S. Let $f(v)$ denote the *expected cost* of a path taken by the game starting from $v \in V$ using the strategy S. One can write the following equations for f:

$$f(v) = \begin{cases} 0, \text{ if } v \in P; \\ \mathit{cost}(S(v)) + f(u), \\ \quad \text{if } v \in \mathit{States} - P \text{ and } S(v) = (v,u); \\ \sum_{e \in E} p(e)(\mathit{cost}(e) + f(u)), \\ \quad \text{if } v \in CP \text{ and } e = (v,u). \end{cases} \qquad (12)$$

We believe that this system of equations always has a unique solution for f for a given strategy S, say θ_S. The goal is to search for a strategy S that minimizes θ_S. We have not looked at this minimization problem, and we are not aware of references where this problem has been addressed.

One can show that our test graph and game definitions are closely related to Markov Decision Processes with finite horizon (MDP), see for example [24, 12]. A pair (s, a) from MDP, where s is a state and a is an action, corresponds to an edge from a state vertex of the test graph to a choice point u. The function $f_u(v) = \sum_{(u,v) \in E} p((u, v))$ gives the distribution on the set of states for MDP. As far as we know, usual approaches to strategy or policy optimisations on MDP minimize the expectation and not the cost maximum combined with the probability of winning. An example of expectation is "the expected total reward criterion" [24, Section 4.1.2]. We think that the representation of MDP as a graph simplifies the application of graph algorithms. Using a graph representation is essential in Section 5. This approach was also chosen for example in [9] in the definition of Simple Stochastic Games.

Model-based testing has recently received a lot of attention and there are several projects and model-based testing tools that build upon FSM based testing theory or LTS based testing theory [2, 11, 18, 21, 28]. Typically, goal oriented testing is provided through model-checking support that produces counterexamples that may serve as test sequences. The basic assumption in those cases is that the system under consideration is either deterministic, or that the nondeterminism can be resolved by the tester (i.e. the systems are made deterministic from the testers point of view). Efficient algorithms for generation of goal oriented test trees or game strategies is not supported in any tools we know of and has, as far as we know, not been addressed in general in the context of testing. This is an open field of research that we are currently investigating.

8. ACKNOWLEDGEMENTS

We thank Andreas Blass and Yuri Gurevich for many valuable comments and discussions during the development of the algorithms presented in this paper.

9. 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] 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.

[3] AsmL. URL: http://research.microsoft.com/fse/AsmL/.

[4] 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.

[5] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Elsevier Journal of Systems and Software*, 65(3):199–208, 2003.

[6] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, second edition, 1990.

[7] D. Box. Code name Indigo: A guide to developing and running connected systems with Indigo. *MSDN magazine*, 2003. URL: http://msdn.microsoft.com/msdnmag/.

[8] 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.

[9] A. Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[11] J. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming - Special Issue on COST247, Verification and Validation Methods for Formal Descriptions*, 29(1-2):123–146, 1997.

[12] J. Filar and K. Vrieze. *Competitive Markov decision processes*. Springer, Berlin, 1996.

[13] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[14] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[15] 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.

[16] 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.

[17] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[18] 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.

[19] N. Immerman. Number of quantifiers is better than number of tape cells. *Journal of Computer and System Sciences*, 22(3):384–406, 1981.

[20] N. D. Jones and W. T. Laaser. Complete problems for deterministic polynomial time. *Theoretical Computer Science*, 3(1):105–117, 1976.

[21] 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.

[22] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1123, Berlin, Aug 1996.

[23] K. Mehlhorn and G. Schäfer. A heuristic for Dijkstra's algorithm with many targets and its use in weighted matching algorithms. In Meyer, editor, *Algorithms - ESA 2001: 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings*, volume 2161 of *LNCS*, pages 242–253. Springer, 2001.

[24] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, New York, 1994.

[25] H. Robinson. Model-based testing home page. URL: http://www.geocities.com/model_based_testing/.

[26] D. P. Sidhu and T.-K. Leung. Formal methods for protocol testing: A detailed study. 15(4):413–426, April 1989.

[27] 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.

[28] 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.

[29] W. Yi and K. G. Larsen. Testing probabilistic and nondeterministic processes. In *Testing and Verification XII*, pages 347–61. North Holland, 1992.

[30] F. Zhang and T. Cheung. Optimal transfer trees and distinguishing trees for testing observable nondeterministic finite-state machines. *IEEE Transactions on Software Engineering*, 29(1):1–14, 2003.

Appendix: Blackjack rules

Blackjack is played with 1 to 6 decks of 52 cards each. The values of the cards correspond to their numerical value from 2-10. All face cards (Jack, Queen, King) count 10 and the Ace either 1 or 11, as the holders desires. A score with an ace valued as 11 is named soft-hand. The color of the cards does not have any effect. The goal of the game is to reach a score, which is the sum of the cards, as high as possible but not more than 21. A Blackjack (Ace and a card whose value is 10) beats all other combination of cards. If the final sum is higher than the sum of the dealer, the player gets a play-off of 1:1 of his initial stake. If the players combination is Blackjack, the play-off is 3:2 of the initial stake. If the sum of the dealer is higher, the player loses his bet. If the sum is equal, then nobody wins. If the player holds a score of 22 or more, he busted and thus he loses his bet immediately. If the dealer busts, the players wins independently of his final score. Blackjack can be played from one to seven players against one dealer. The dealer shuffles the cards. Now the players must place their bets. Then each player and the dealer receives one card. The cards all lie face up. Thereafter the player receive a second card. The player now can continue to 'buy' further cards, one by one, until he believes that he is near enough to 21. If the player believes to have reached a score high enough he must to signal the dealer to 'stay', which means not to call for any further cards.