Deep Random Search for Efficient Model Checking of Timed Automata

R. Grosu[1], X. Huang[1], S.A. Smolka[1], W. Tan[1] and S. Tripakis[2]

[1]Dept. of CS, Stony Brook Univ., Stony Brook, NY 11794, USA
E-mail: {grosu,xhuang,sas,wktan}@cs.sunysb.edu

[2]Verimag, Centre Equation, 38610 Gieres, France
E-mail: {tripakis}@imag.fr

Abstract. We present DRS (*Deep Random Search*), a new Las Vegas algorithm for model checking safety properties of timed automata. DRS explores the state space of the simulation graph of a timed automaton by performing random walks up to a prescribed depth. Nodes along these walks are then used to construct a random fringe, which is the starting point of additional deep random walks. The DRS algorithm is complete, and optimal to within a specified depth increment. Experimental results show that it is able to find extremely deep counter-examples for a number of benchmarks, outperforming Open-Kronos and UPPAAL in the process.

1 Introduction

The goal of this paper is to demonstrate the effectiveness of random search in the model checking of timed automata (TA). To this end, we present the *Deep Random Search* (DRS) algorithm for checking safety properties of TA. DRS is an iterative-deepening, deep-random-walk, random-fringe-backtracking Las-Vegas algorithm. By "deep random walk" we mean that in any state of a random walk, DRS always chooses a random *non-visited* child, as long as such a state exists. By "random fringe backtracking" we mean that the algorithm does not limit backtracking to predecessors; rather it randomly selects a node from the fringe as the starting point for a deep random walk. This strategy removes much of the bias towards the initial state of the search space. We now discuss the algorithm in more detail, highlighting its main features.

- The DRS algorithm operates on *simulation graphs*, an efficient, symbolic representation of timed automata that can be generated on-the-fly [5, 12]. A node of a simulation graph is a symbolic state comprising a finite set of regions all having the same discrete state. Although, in the worst case, a simulation graph can be exponentially large in the size of the underlying TA, in practice, it is orders of magnitude smaller than the region graph.
- DRS is a Las Vegas algorithm, i.e. a randomized algorithm that always produces the correct answer but whose running time is a random variable. The quintessential Las Vegas algorithm is randomized quick sort, which chooses its pivot element randomly and consequently runs in expected time $O(n \log n)$ for *all* input of length n. As explained below, DRS uses iterative deepening to perform a *complete*, albeit random, search of the state space under investigation, thereby qualifying it for its Las Vegas status.
- DRS performs *deep random search* by taking random walks that are as *deep as possible*: they reach a leaf node, a prescribed *cutoff depth*, or a node whose children were already visited (a *closed* node). A node with at least

two unvisited children that is encountered along such a walk it is added to the *fringe*. A closed node is deleted from the fringe. When a deep random walk terminates, a node is picked *at random* from the fringe to commence a new deep random walk. This process continues until the fringe is empty, thereby ensuring completeness up to the cutoff value.
- `DRS` allows the user to initialize the fringe by taking *walks* initial deep random walks, where *walks* ranges between 1 and the number of children of the initial state. Parameter *walks*, in combination with the cutoff value, gives the user control over both the breadth and depth of the random search performed by `DRS`. Should the user have a priori knowledge about the "shape" (density and length) of the execution space and potential counter-examples, then these parameters can be used to fine-tune `DRS`'s performance accordingly.
- Iterative deepening is realized by repeating the deep-random-search process with a new cutoff value equal to that of the old cutoff plus a prescribed *increment*. For an increment of one, `DRS` is *optimal* [16] in the sense that it always finds the shortest counter-example, should one exist. Otherwise, it is optimal up to the value of the increment.

Our experimental results show that for all benchmarks having a counter-example, `DRS` consistently outperforms the Open-Kronos [5] and UPPAAL [12] model checkers. Otherwise, its performance is consistent with that of Open-Kronos. The benchmarks were chosen to exhibit a wide range of counter-examples, with depth from 6 to 13,376. Open-Kronos performs traditional depth-first on simulation graphs. UPPAAL uses Difference Bounded Matrices, Minimal Constraint Representation and Clock Difference Diagrams to symbolically represent the state space, and allows the user to choose between breadth-first and depth-first search.

In related work, a number of researchers have investigated the use of random search (i.e. random walk) in model checking and reported on its benefits, including [14, 7, 18, 10, 15]. To the best of our knowledge, `DRS` is the first *complete* Las Vegas algorithm to be proposed for the problem.[1]

The rest of the paper is organized as follows. Sections 2 and 3 review the theory of timed automata and simulation graphs. Section 4 presents our `DRS` algorithm, while Section 5 discusses our experimental results. Section 6 offers our concluding remarks.

2 Timed Büchi Automata

In this section we define Timed Büchi automata, a real-time extension of classical Büchi automata that will serve as our formal model of real-time systems. We begin with some preliminary definitions. Let N denote the natural numbers, R the non-negative real numbers, and let $\mathcal{X}$ be a finite set of variables taking values in R. In our definition of a Timed Büchi automaton to follow, $\mathcal{X}$ will be a finite

[1] Randomized SAT solvers for bounded model checking [3] and the algorithm of [10] are heuristics-based guided search algorithms in which randomization plays a secondary role; e.g., to break ties among alternatives with the same cost. In contrast, randomization is the primary algorithmic technique utilized by `DRS`.

set of *clock* variables. An $\mathcal{X}$*-valuation* is a function $\mathbf{v} : \mathcal{X} \to \mathsf{R}$ that assigns to each variable in $\mathcal{X}$ a value in R. $\mathbf{0}$ denotes the valuation assigning 0 to all variables in $\mathcal{X}$. Given a valuation $\mathbf{v}$ and $\delta \in \mathsf{R}$, $\mathbf{v}+\delta$ is defined to be the valuation $\mathbf{v}'$ such that $\mathbf{v}'(x) = \mathbf{v}(x) + \delta$ for all $x \in \mathcal{X}$. Given a valuation $\mathbf{v}$ and $X \subseteq \mathcal{X}$, $\mathbf{v}[X := 0]$ is defined to be the valuation $\mathbf{v}'$ such that $\mathbf{v}'(x) = 0$ if $x \in X$ and $\mathbf{v}'(x) = \mathbf{v}(x)$ otherwise.

An *atomic constraint* on $\mathcal{X}$ is a constraint of the form $x\#c$, where $x, y \in \mathcal{X}$, $c \in \mathsf{N}$ and $\# \in \{<, \leq, \geq, >\}$. A valuation $\mathbf{v}$ *satisfies* an atomic constraint α, denoted $\mathbf{v} \models \alpha$, if substituting the values of the clocks in the constraint yields a valid inequality. For example, $\mathbf{v} \models x \leq 5$ iff $\mathbf{v}(x) \leq 5$. A conjunction of atomic constraints defines a set of $\mathcal{X}$-valuations, called an $\mathcal{X}$*-zone*. For example, $x \leq 5 \wedge y > 3$ defines the set of all valuations $\mathbf{v}$ such that $\mathbf{v}(x) \leq 5 \wedge \mathbf{v}(y) > 3$.[2]

Definition 1 (Timed Büchi Automaton [1]). *A* timed Büchi automaton *(TBA) is a six-tuple* $T = (\mathcal{X}, Q, q_0, E, \mathsf{invar}, F)$*, where:*

- $\mathcal{X}$ *is a finite set of* clocks*.*
- Q *is a finite set of* discrete states*,* $q_0 \in Q$ *being the* initial *discrete state.*
- $F \subseteq Q$ *is a finite set of* accepting states*.*
- E *is a finite set of* edges *of the form* $e = (q, \zeta, X, q')$*, where* $q, q' \in Q$ *are the* source *and* target *discrete states,* ζ *is an* $\mathcal{X}$*-zone, called the* guard *of* e*, and* $X \subseteq \mathcal{X}$ *is a set of clocks to be* reset *upon taking the edge.*
- invar*, the* invariant *of* q*, is a function that associates an* $\mathcal{X}$*-zone with each discrete state* q*.*

Given an edge $e = (q, \zeta, X, q')$, we write $\mathsf{source}(e)$, $\mathsf{target}(e)$, $\mathsf{guard}(e)$ and $\mathsf{reset}(e)$ for q, q', ζ and X, respectively. Given a discrete state q, we write $\mathsf{in}(q)$ (resp. $\mathsf{out}(q)$) for the set of edges of the form $(\_, \_, \_, q)$ (resp. $(q, \_, \_, \_)$). We assume that for each $e \in \mathsf{out}(q)$, $\mathsf{guard}(e) \subseteq \mathsf{invar}(q)$.

A *state of* A is a pair $s = (q, \mathbf{v})$, where $q \in Q$ and $\mathbf{v} \in \mathsf{invar}(q)$. We write $\mathsf{discrete}(s)$ to denote q. The *initial state* of A is $s_0 = (q_0, \mathbf{0})$.

An edge $e = (q_1, \zeta, X, q_2)$ can be seen as a (partial) function on states. Given a state $s = (q_1, \mathbf{v})$ such that $\mathbf{v} \in \zeta$ and $\mathbf{v}[X := 0] \in \mathsf{invar}(q_2)$, $e(s)$ is defined to be the state $s' = (q_2, \mathbf{v}[X := 0])$. Whenever $e(s)$ is defined, we say that a *discrete transition* can be taken from s to s'.

A real number $\delta \in \mathsf{R}$ can also be seen as a (partial) function on states. Given a state $s = (q, \mathbf{v})$, if $\mathbf{v} + \delta \in \mathsf{invar}(q)$ then $\delta(s)$ is defined to be the state $s' = (q, \mathbf{v} + \delta)$; otherwise $\delta(s)$ is undefined. Whenever $\delta(s)$ is defined, we say that a *time transition* can be taken from s to s'.

An infinite sequence of pairs $(\delta_0, e_0), (\delta_1, e_1), ...$, where for all $i = 0, 1, ...$, $\delta_i \in \mathsf{R}$ and $e_i \in E$, defines a *run of* A *starting at state* s, if s is a state of A and the sequence of states $s_0 = s$, $s_{i+1} = e_i(\delta_i(s_i))$ is defined for all $i \geq 0$. The run is called *accepting* if there exists an infinite set of indices $J \subseteq \mathsf{N}$, such that for all $i \in J$, $\mathsf{discrete}(s_i) \in F$. The run is called *zeno* if the sequence

[2] Zones are particularly interesting since they can be represented using $O(n^2)$ space-efficient data-structures such as *difference-bound matrices* [6], where n is the number of clocks. Standard operations on these data structures are also time-efficient; e.g., intersection in $O(n^2)$, test for emptiness in $O(n^3)$.

$\delta_0, \delta_0+\delta_1, \delta_0+\delta_1+\delta_2, \ldots$ *converges*, that is, if $\exists \delta \in \mathsf{R}, \forall k = 0, 1, \ldots, \Sigma_{i=0,\ldots,k}\delta_i < \delta$. Otherwise, the run is called *non-zeno*.

Example 1. Consider the two TBA of Figure 1. Circles represent discrete states, double circles represent accepting states, and arrows represent edges. Labels a, b, c refer to edges. A run of A_1 starting at state $(q_0, \mathbf{0})$ is $(0.5, a)$, $(0.25, a)$, $(0.125, a), \ldots$; this run is zeno. In fact, any run of A_1 taking a-transitions forever is zeno. On the other hand, the run $(0, b), (1, c), (1, c), \cdots$ of A_1 is non-zeno. Finally, every accepting run of A_2 is non-zeno.

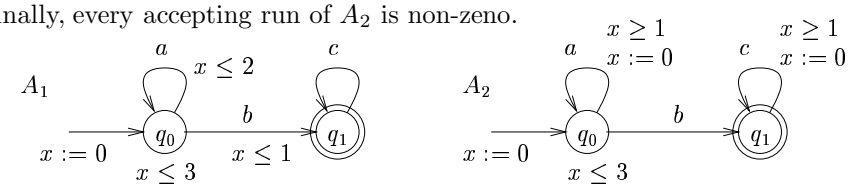

Fig. 1. A TBA with zeno runs (left) and a strongly non-zeno TBA (right).

Definition 2 (Language and emptiness problem for TBA). *The* language *of A, denoted $Lang(A)$, is defined to be the set of all non-zeno accepting runs of A starting at the initial state s_0. The* emptiness problem *for A is to check whether $Lang(A) = \emptyset$.*

The emptiness problem for TBA is known to be PSPACE-complete [1]. More precisely, the worst-case complexity of the problem is linear in the number of discrete states of the automaton, exponential in the number of clocks, and exponential in the encoding of the constants appearing in guards or invariants. This worst-case complexity is inherent to the problem: as shown in [4], both the number of clocks and the magnitude of the constants render PSPACE-hardness independently of each other.

Definition 3 (Strong non-zenoness). *A TBA A is called* strongly non-zeno *if all accepting runs starting at the initial state of A are non-zeno.*

A *structural loop* of a TBA A is a sequence of distinct edges $e_1 \cdots e_m$ such that $\mathsf{target}(e_i) = \mathsf{source}(e_{i+1})$, for all $i = 1, \ldots, m$ (the addition $i + 1$ is modulo m). We say that the structural loop is *accepting* if there exists an index $1 \leq i \leq m$ such that $\mathsf{target}(e_i)$ is an accepting state. We say that the structural loop *spends time* if there exists a clock x of A and indices $0 \leq i, j \leq m$ such that:

1. x is reset in step i, that is, $x \in \mathsf{reset}(e_i)$, and
2. x is bounded from below in step j, that is, $(x < 1) \cap \mathsf{guard}(e_j) = \emptyset$.

Definition 4 (Structural non-zenoness). *We say that a TBA A is* structurally non-zeno *if every accepting structural loop of A spends time.*

For example, in Figure 1, automaton A_1 is not structurally non-zeno, while automaton A_2 is. A_2 would not be structurally non-zeno if any of the guards $x \geq 1$ were missing.

Lemma 1 ([17]). *If A is structurally non-zeno then A is strongly non-zeno.*

Theorem 1 ([17]). *Any TBA A can be transformed into a strongly non-zeno TBA A', such that: (1) A' has one clock more than A and (2) $Lang(A) = \emptyset$ iff $Lang(A') = \emptyset$.*

3 Simulation Graphs

Simulation graphs were introduced in [5] as a technique for checking reachability in timed automata. In [17, 2], it is shown how simulation graphs can also be used to check TBA emptiness. We summarize these results in this section.

Consider a TBA $A = (\mathcal{X}, Q, q_0, E, \mathsf{invar}, F)$. A *symbolic state* S of A is a finite set of *regions* [1] $r_i = (q, \zeta_i)$, $1 \leq i \leq k$, all associated with the same discrete state $q \in Q$. We also denote S as (q, ζ), where $\zeta = \cup_i \zeta_i$.[3] Given an edge $e \in E$, let $e(S)$ be the set of all regions r' for which there exists $r \in S$ such that r can reach r' by a transition labeled e in the region graph. Similarly, let $\epsilon(S)$ be the set of all regions r' for which there exists $r \in S$ such that r can reach r' by a (possible empty) sequence of time-passing transitions in the region graph (thus, $S \subseteq \epsilon(S)$). Then, we define $\mathsf{post}(S, e) = \epsilon(e(S))$.

Definition 5 (Simulation graph). *The* simulation graph *of a TBA A, denoted $SG(A)$, is the graph whose set of nodes $\mathcal{S}$ is the least set of non-empty symbolic states of A, such that:*

1. *$\epsilon((q_0, \mathbf{0})) \in \mathcal{S}$ is the initial node of $SG(A)$), and*
2. *if $e \in E$, $S \in \mathcal{S}$ and $S' = \mathsf{post}(S, e)$ is non-empty, then $S' \in \mathcal{S}$.*

$SG(A)$ has an edge $S \xrightarrow{e} S'$ iff $S, S' \in \mathcal{S}$ and $S' = \mathsf{post}(S, e)$.

An example of a TBA and its simulation graph is given in Figure 2. The simulation graph was automatically generated using the KRONOS [5] tool.

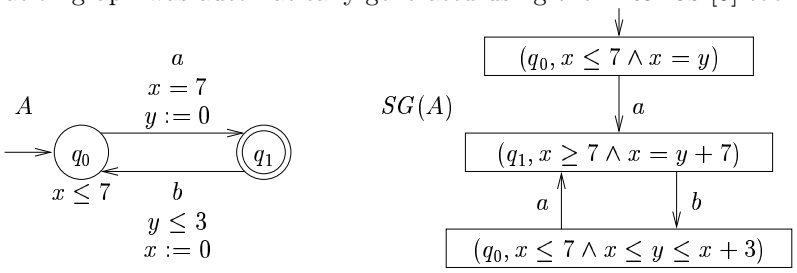

Fig. 2. A TBA and its simulation graph.

A node $(q, \zeta) \in \mathcal{S}$ is *accepting* if $q \in F$. Let $\mathcal{F}$ and Δ be the set of accepting nodes and the set of edges of $SG(A)$, respectively. The simulation graph $SG(A) = (E, \mathcal{S}, \mathcal{S}_0, \Delta, \mathcal{F})$ defines a discrete Büchi automaton (BA) over the input alphabet E and symbolic states $\mathcal{S}$.

A sequence $\sigma = S_0 \xrightarrow{e_0} S_1 \xrightarrow{e_1}, ...,$ where $S_0 \in \mathcal{S}_0$ and for all $i \geq 0$, $S_i \xrightarrow{e_i} S_{i+1} \in \Delta$ is called an *infinite run* of $SG(A)$ if the sequence is infinite and a *finite run* otherwise. An infinite run is called *accepting* if there exists an infinite set of indices $J \subseteq \mathsf{N}$, such that for all $i \in J$, $S_i \in \mathcal{F}$. We say that σ is ultimately periodic if there exist $i \geq 0$, $l \geq 1$ such that for all $j \geq 0$, $S_{i+j} = S_{i+j \bmod l}$. This means that σ consists of a finite prefix $S_0 \xrightarrow{e_0} \cdots S_{i-1} \xrightarrow{e_{i-1}}$, followed by the

[3] A union of regions is generally not convex. In practice, tools such as Kronos work with symbolic states that can be represented by zones; i.e., such that $\cup_i \zeta_i$ is convex.

"infinite unfolding" of a *cycle* $S_i \xrightarrow{e_i} \cdots S_{i+l-1} \xrightarrow{e_{i+l-1}} S_i$. The cycle is called *simple* if for all $0 \leq j \neq k < l$, $S_{i+j} \neq S_{i+k}$, that is, the cycle does not visit the same node twice. In the following, we refer to such a reachable simple cycle as a *lasso* and say that the lasso is accepting if its simple cycle contains an accepting node.

Theorem 2 ([17]). *Let A be a strongly non-zeno TBA. $Lang(A) \neq \emptyset$ iff there is an accepting lasso in the simulation graph of A.*

4 The Deep-Random-Search Algorithm

Let A be a strongly non-zeno TBA and let $\mathcal{S} = SG(A)$ be A's simulation graph; as shown in Section 3, $\mathcal{S}$ exists and there is an efficient procedure for generating it from A. Moreover, $\mathcal{S}$ is a Büchi automaton (BA). Now let φ be a real-time property expressed in a logic for timed automata, e.g., TECTL$^*_\exists$ [2]. Since the formulas of this logic are built up from timed automata, we can construct, as shown in [2], a corresponding BA $T = SG(A_{\neg\varphi})$, the simulation graph of the snz TBA $A_{\neg\varphi}$. The TECTL$^*_\exists$model-checking problem $A \models \varphi$ is then naturally defined in terms of the TBA emptiness problem for $S \times T$.

If φ is a *safety property*, then T has an associated deterministic finite automaton $\text{pref}(T)$ that recognizes all finite trajectories violating the property [11]. As a consequence, the model-checking problem for safety properties can be reduced to a reachability (of accepting states) problem on the product automaton $B = \text{fin}(S) \times \text{pref}(T)$, where $\text{fin}(S)$ is the finite automaton recognizing all finite trajectories of S.

Our model checker for timed automata applies the deep-random-search (`DRS`) algorithm described below to the finite automaton B. As discussed in Section 1, `DRS` is an iterative-deepening, deep-random-walk, random-fringe-backtracking Las-Vegas algorithm. By "deep random walk" we mean that in any state of a random walk, `DRS` always chooses a random non-visited child (immediate successor) state, as long as such a state exists. By "random fringe backtracking" we mean that the algorithm does not limit backtracking to predecessors; rather it randomly selects a node from the fringe as the starting point for a deep random walk. This strategy removes much of the bias towards the initial state of the search space. We assume that B is given as the triple (`initState, Next-Child, Acc`) where `initState` is the initial state, `Next-Child` is an iterator function for the immediate successors of a state of B, and `Acc` is a predicate defining the accepting states of B.

To fine tune the breath and the depth of the search, `DRS` inputs three additional parameters: `walks`, `cutoff` and `increment`. The first of these is the number of initial deep random walks taken by the algorithm from the root, which is always constrained to be greater than one and less then the number of children of `initState`. While not affecting completeness, this parameter determines the initial fringe, and therefore influences the way the computation tree grows. The second parameter bounds the depth of the search; thus, it affects completeness. To obtain a complete algorithm, `cutoff` has to be set to infinity. The third parameter is the iterative-deepening increment. While not affecting completeness,

```
State initState; State Next-Child(State); Bool Acc(State);
NodeSet fringe=empty; StateSet generated=empty; Bool done=false;

void DRS(Int increment) {
  for (Int co=increment; (!done && co≤cutoff); co+=increment); {
    done=true; generated=empty; fringe=empty;
    Bounded-DRS(co); }
  exit("no counter-example"); }

void Bounded-DRS(Int cutoff) {
  if (Acc(initState)) exit("counter-example", initState);
  Insert(generated, initState);
  NodeSet children = Nonaccepting-Interior-Children((initState,1));
  for (Int i=1; (children!=empty && i≤walks); i++){
    Node node = Random-Remove(children);
    Insert(generated, node.state); Insert(fringe, node);
    Random-Walk(node, cutoff); }
  while (fringe != empty) {
    node=Random(fringe);
    Random-Walk(node, cutoff); }
  return;}

void Random-Walk(Node node, Int cutoff) {
  Node next=node;
  while (next.depth < cutoff) {
    NodeSet children=Nonaccepting-Interior-Children(next) ;
    if (|children|≤1) {Remove(fringe, next); if (|children|==0) return;}
    next=Random(children);
    Insert(generated, next.state); Insert(fringe, next); }
  Remove(fringe, next); done=false; return;}

Node Set Nonaccepting-Interior-Children(Node nd) {
  NodeSet children=empty;
  for(State nx=Next-Child(nd.state);nx!=Null;nx=Next-Child(nd.state)){
    if (!In(generated, nx)) {
      if (Acc(nx)) exit("counter-example", nx);
      if (!Leaf(nx)) Insert(children, (nx, nd.depth+1));} } }
  return children;}
```

Fig. 3. DRS model-checking algorithm.

this parameter may affect optimality. Setting `increment` to one ensures the algorithm is optimal. Note, however, that (theoretical) optimality may lead to poor performance for deep counter-examples, as the search has to explore all the states in the tree above the counter-example.

The pseudo-code for `DRS` is shown in Figure 3. It uses the following three global variables: `generated`, `fringe` and `done`. The first of these is the set of so-far-generated states and is used to ensure that no state is visited more than once. The second variable is the set of generated states with unexplored successors, together with their depth. We call a state together with its depth a *node*. The third variable is a flag which is true when no random walk has been cutoff and therefore the entire search space has been explored.

Procedure `DRS` has an iterative-deepening for-loop. In each iteration, it initializes the global variables, it increments the cutoff depth and then calls proce-

dure `Bounded-DRS`. This procedure returns only if no counter-example was found. Moreover, if no random walk was cut off, upon return from `Bounded-DRS`, the flag `done` is still true, signaling that the entire state space of B has been explored without finding a counter-example.

Procedure `Bounded-DRS` performs a complete search of the transition graph of B up to the cutoff depth. The procedure first checks whether the initial state is accepting in which case it exits and signals that it has found a counter-example. Otherwise, it initializes the fringe by taking the number of deep walks specified by the parameter `walks`. Each such walk starts from a different child of the initial state. As long as the fringe is not empty, the procedure then repeatedly starts deep random walks from a random node in the fringe and up to the cutoff depth, by calling `Random-Walk`.

Procedure `Random-Walk` traverses a deep random path in the computation tree of B. For each node along the path, if first obtains the set of all the non-accepting, non-generated, interior children of the node. If this set is empty, or if it contains only one node, the current node can be safely removed from the fringe, as all its successors have been (or are in the process of being) explored. Moreover, if the set is empty, the walk cannot be continued and the procedure returns. Otherwise, it randomly picks one of the children, inserts it in the generated set and in the fringe and continues from this node. The procedure is guaranteed to stop when the cutoff value is reached. In this case, the cutoff node is removed from the fringe and `done` is set to false.

Procedure `Nonaccepting-Interior-Children` uses the iterator `Next-Child` of B to construct the set of interesting children of the current node. A child state of the current node's state is inserted (together with its depth) in this set only if the state was not previously generated, is non-accepting, and has at least one enabled successor (it is not a leaf).

Theorem 3 (Correctness & completeness). *Given a timed automaton A and safety property φ, DRS-MC returns a counter-example if and only if $A \not\models \varphi$.*

Proof. The proof follows from Theorems 1, 2 and the fact that the `DRS` model-checking algorithm is complete.

Theorem 4 (Complexity). *Let $B = \mathrm{fin}(S) \times \mathrm{pref}(T)$ be the finite automaton discussed above. Then* `DRS` *uses $O(|B|)$ time and space, where $|B|$ is the number of states in B.*

Theorem 5 (Optimality). *Let d be the smallest depth of an accepting state of B. Then the depth of a counter-example returned by* `DRS` *is never greater than d provided* `increment` *is set to one.*

5 Experimental Results

We have implemented the `DRS` model-checking algorithm as an extension to the Open-Kronos model checker for timed automata [17]. Open-Kronos takes as input a system of extended timed automata and a boolean expression defining the accepting states of the automata. The input is translated into a C program

which is compiled and linked to the Profounder, a tool that performs on-the-fly generation of the simulation graph of the input TA model and applies standard depth-first search for reachability analysis.

To assess the performance and scalability of `DRS`, we compared its performance to Open-Kronos and UPPAAL (3.4.11) on the following real-time model-checking benchmarks: the *Fischer Real-Time Mutual-Exclusion Protocol*, the *Philips Audio Protocol*, and the *Bang & Olufsen Audio/Video Protocol*. All reported results (Tables 1-4) were obtained on a PC equipped with an Athlon 2600+ MHz processor and 1GB RAM running Linux 2.6.5 (Fedora Core 2).

In the tables, the meaning of the column headings is the following: `proc` is the number of processes; `sender` is the number of senders (Tables 3 and 4); `time` is given in `seconds`; `states` is the number of visited states—for `DRS`, this is the size of the set *generated*; `depth` is the depth of the accepting state found by the model checker; and `oom` means the model checker ran out of memory. The statistics provided for `DRS` are averages obtained over a representative number of runs of the algorithm. Because UPPAAL does not provide the number of visited states, path depth, etc., only its execution time is given here.

	Open-Kronos			DRS			UPPAAL
proc	time	states	depth	time	states	depth	time
2	0.038	63	44	0.003	20	6	0.021
4	2.968	1227	1166	0.006	67	28	0.041
8	13.20	35409	2048	0.082	216	211	1.280
12	204	332253	2048	0.512	386	374	18.61
16	>12hrs	?	?	0.906	238	222	223 (oom)

Table 1. Mutual exclusion for Fischer protocol (buggy version).

	Open-Kronos		DRS	UPPAAL
proc	time	states	time	time
2	0.004	203	0.011	0.02
3	0.386	24949	0.513	0.03
4	943	3842501	1388	0.14
5	4hrs	oom	oom	2.01
6	4hrs	oom	oom	124
7	4hrs	oom	oom	>5hrs

Table 2. Mutual exclusion for Fischer protocol (correct version).

The Fischer protocol uses timing constraints and a shared variable to ensure mutual exclusion among processes competing for access to a critical section [13]. Table 1 contains the results for checking mutual exclusion (a safety property) on a buggy version of the protocol. As the results indicate, `DRS` consistently outperforms Open-Kronos and UPPAAL, thereby illustrating the power of deep random search in finding counter-examples. Table 2 contains the mutual-exclusion results for the correct version of the protocol. In this case, `DRS`, like Open-Kronos, must perform a complete search of the state space and the performance of the two model checkers is similar. UPPAAL's performance, on the other hand, is superior

to that of Open-Kronos and DRS, although it too struggles when the number of processes reaches 7.

	Open-Kronos			DRS			Uppaal
sender	time	states	depth	time	states	depth	time
1	0.004	72	71	0.003	16	12	0.026
4	3.259	46263	2048	0.007	30	26	0.041
8	422.2	1026446	2048	0.041	93	26	0.158
12	>12hrs	?	?	0.736	375	42	0.802
24	>12hrs	?	?	0.020	41	17	39.095
28	>12hrs	?	?	0.033	50	22	107 (oom)

Table 3. Results for the Phillips audio protocol.

The purpose of the Phillips audio protocol is to exchange control information between audio components using the Manchester encoding [9]. The correctness of the encoding relies on timing delays between signals. The protocol was designed to satisfy the following safety property: communication between components should be reliable, with a tolerance of ±5% on the timing. However, the protocol is faulty and our results are given in Table 3 for a varying number of sender components. For this benchmark, DRS consistently outperforms both Open-Kronos and Uppaal.

	Open-Kronos			DRS			Uppaal
sender	time	states	depth	time	states	depth	time
2	0.226	1285	1284	0.034	1659	1657	0.174
3	35.161	1135817	1997	10.76	166113	2318	1.050
4	53.532	1130669	1608	50.554	617760	2972	10.1
5	1200	oom	-	10 min	6769520	4734	2 min
6	1200	oom	-	37 min	30316978	13376	12 min (oom)

Table 4. Results for the B&O audio/video protocol.

The Bang & Olufsen audio/video protocol was designed to transmit messages between audio/video components over a single bus. Its behavior is highly timing dependent. The protocol is intended to satisfy the following safety property: whenever a frame has been sent, the transmitted frame must be intact, and other senders must not have discovered a collision [8]. The results of Table 4 show once again that deep random search is superior to depth-first search (Open-Kronos) in finding deep counter-examples. DRS's performance is similar to that of Uppaal on this benchmark. For 6 senders, the results reported for DRS are those for one out of 20 executions of the model checker; the other 19 ran out of memory.

6 Conclusions

We have presented the DRS deep-random-search algorithm for model checking timed automata. DRS performs random walks up to a prescribed depth within the TA's simulation graph. Nodes along these walks are then used to construct a random fringe, which is the starting point of additional deep random walks. DRS is complete, and optimal to within a specified depth increment. Our experimental results show that it is able to find extremely deep counter-examples while consistently outperforming the Open-Kronos and UPPAAL model checkers. Our DRS algorithm is not restricted to timed automata; it may be beneficially applied to the model checking of safety properties of any concurrent system.

A version of DRS that is more in line with classic depth-first or breadth-first search, would put all the non-accepting, non-generated interior children (except for the one randomly selected) in the fringe and not the node itself. Intuitively, this version should perform less work, as it explores the children of a node only once. We have implemented this version too, but found that it performed worse in terms of finding counter-examples. The reason for the performance degradation may be the fact that more nodes are inserted in the fringe with each deep random walk and therefore the chance of selecting the right deep candidate node may decrease, at least for the examples that we have tested.

The expected time complexity of DRS is related to the random variable X, the number of states visited before an accepting state is found. Getting a closed-form expression for the mean and variance of X is difficult due to the intricate interdependence between the random walks taken by the algorithm. This is a subject for future work. Experimental results for a guided-search algorithm, where randomization is used to select a successor among the first n elements in a priority queue, showed that X follows a normal distribution [10]. Increasing n was shown to increase both the variance and the mean of X. Randomization improved the search performance because the probability of observing a small value of X increased logarithmically with the variance, provided the mean remained unchanged.

We also plan to investigate how to extend the deep-random-search technique to liveness properties. The main issue here is deciding when a deep random walk has formed a lasso. It is not enough to terminate such a walk when a previously visited state is re-encountered; rather one must correctly distinguish cross-edges from back-edges in the simulation graph. This would probably require storing parent edges, which are also useful in determining the path from the initial state to the accepting state.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model checking for real-time systems. In *18th IEEE Real-Time Systems Symposium (RTSS'97), San Francisco, CA*, pages 25–34. IEEE, December 1997.

3. E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *In Formal Methods in System Design*, 19(1):7–34, July 2001.
4. C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. In *CAV'91*, LNCS 575. Springer, 1991.
5. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.
6. D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer–Verlag, 1989.
7. P. Haslum. Model checking by random walk. In *Proc. of 1999 ECSEL Workshop*, 1999.
8. K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Automated analysis of an audio control protocol. In *Proc. of 18th IEEE Real-Time Systems Symposium*, pages 2–13, San Francisco, California, USA, December 1997.
9. P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In *Proc. of CAV 95*, number 939 in Lecture Notes in Computer Science, pages 381–394. Springer-Verlag, 1995.
10. M. Jones and E. Mercer. Explicit state model checking with Hopper. In *Proceedings of the 11th SPIN Workshop*. Volume 2989, Lecture Notes in Computer Science, Springer-Verlag, 2004.
11. O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
12. K. Larsen, P. Petterson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1/2), October 1997.
13. K. G. Larsen, P. Pettersson, and W. Yi. Model checking for real-time systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in Lecture Notes in Computer Science, pages 62–88, August 1995.
14. M. Mihail and C. H. Papadimitriou. On the random walk method for protocol testing. In *6th International Conference on Computer Aided Verification (CAV)*, pages 132–141. Springer, LNCS 818, 1994.
15. R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing random walk state space exploration. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 98–105. ACM Press, 2005.
16. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002.
17. S. Tripakis, S. Yovine, and A. Bouajjani. Checking timed Büchi automata emptiness efficiently. *Formal Methods in System Design*, 26(3):267–292, May 2005.
18. E. Tronci, G., D. Penna, B. Intrigila, and M. Venturini. A probabilistic approach to automatic verification of concurrent systems. In *Proc. of 8th IEEE Asia-Pacific Software Engineering Conference (APSEC)*, 2001.