Checking Finite Traces using Alternating Automata *

Bernd Finkbeiner and Henny Sipma
Computer Science Department, Stanford University
Stanford, CA. 94305

Abstract. Alternating automata have been commonly used as a basis for static verification of reactive systems. In this paper we show how alternating automata can be used in runtime verification. We present three algorithms to check at runtime whether a reactive program satisfies a temporal specification, expressed by a linear-time temporal logic formula. The three methods start from the same alternating automaton but traverse the automaton in different ways: depth-first, breadth-first, and backwards, respectively. We then show how an extension of these algorithms, that collects statistical data while verifying the execution trace, can be used for a more detailed analysis of the runtime behavior. All three methods have been implemented and experimental results are presented.

1. Introduction

Software model checking is hard, and in the majority of cases infeasible. A practical alternative might be to monitor the running program, and check on the fly whether desired temporal properties hold. Another situation in which runtime monitoring may be the only option is when the program must run in an environment that does not tolerate violations of the specification, but the source code is unavailable for inspection for proprietary reasons or due to outsourcing. Thus there is a need for efficient methods that, given a specification, will check program output or state against this specification.

Alternating automata are an efficient data structure for many problems in the specification and verification of reactive systems. Because of their succinctness they are a convenient way to represent temporal specifications (Vardi, 1996; Vardi, 1997). They also have been commonly used as a basis for static verification (Vardi, 1995).

In this paper we show how alternating automata can be used in the verification of individual traces, also known as runtime verification. In our approach, alternating automata are used as an intermediate data structure to check whether a program trace satisfies a (future) linear-time temporal specification. We present three algorithms that each take

* This research was supported in part by the National Science Foundation grant CCR-99-00984-001, by ARO grant DAAG55-98-1-0471, by ARO/MURI grant DAAH04-96-1-0341, by ARPA/Army contract DABT63-96-C-0096, and by ARPA/AirForce contracts F33615-00-C-1693 and F33615-99-C-3014.

© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

as input the same automaton, but traverse the automaton in different ways, which allows us to easily vary the runtime characteristics of the runtime checking without affecting its semantics.

The first algorithm uses a depth-first strategy: following the structure of the automaton, it attempts to construct a tree that maps the trace onto an unrolled version of the automaton. This algorithm is the most intuitive and easiest to implement. However, it is generally inefficient for long traces because large parts of the trace may have to be traversed multiple times in the search for a satisfying state.

The second algorithm overcomes this problem by using a breadth-first strategy, allowing it to traverse the trace only once. At every position of the trace it keeps track of all possible sets of automaton nodes that are consistent with the prefix of the trace seen so far. The disadvantage of this algorithm is that it may generate an exponential number of sets of nodes at each position in the trace, making this algorithm unattractive for larger formulas.

The third algorithm is in general the most efficient: by traversing the trace backward it avoids the search for a satisfying mapping between trace states and automaton nodes. Instead, starting from the last state in the trace it simply evaluates satisfaction of each state for all automaton nodes, using the known values of the successor states. With both time and space complexity linear in the size of the specification and the trace, this algorithm is to be preferred when the trace is available off-line.

These algorithms all check whether the trace satisfies the specification. However in many cases a more detailed analysis of the relationship between trace and specification is desirable. For example, the satisfaction of a liveness property over a finite trace is not very meaningful. Instead we may want to know at what rate eventualities are fulfilled in the trace. We show how the trace-checking algorithms can be extended to collect this kind of statistics during their trace traversal, and explain their use with two applications.

The algorithms described so far apply to future temporal formulas only. Extending them for past formulas is straightforward for the depth-first and reverse-traversal algorithms. Extending the breadth-first algorithm is more challenging. We show how, for a restricted class of formulas containing both future and past temporal operators, the breadth-first and reverse-traversal strategies can be combined into one algorithm in a natural way such that the trace is traversed only once, as for the future case.

We implemented the three trace-checking algorithms and report some preliminary experimental results that demonstrate the different runtime behaviors for different formulas and varying trace lengths.

Related Work

Runtime verification has received a growing attention recently. A series of workshops on this topic was initiated (Havelund and Rosu, 2001; Havelund and Rosu, 2002a) as satellite workshops of the International Conference on Computer Aided Verification. Recent work includes the MaC architecture (Lee et al., 1999), which provides both program instrumentation and runtime checking, given a temporal specification. A prototype implementation, Java-Mac (Kim et al., 2001), demonstrates the architecture for Java programs. Commercial runtime verification systems include the Temporal Rover (Drusinsky, 2000), a tool that allows LTL specifications to be embedded in C, C++, Java, Verilog and VHDL programs. Runtime analysis algorithms have also been applied in guiding the Java model checker Java PathFinder developed at NASA (Havelund, 2000). The work presented in this paper was inspired by (Havelund and Roşu, 2001; Roşu and Havelund, 2001).

Outline

The remainder of the paper is organized as follows. In Section 2 we present our specification language of linear time temporal logic (LTL), alternating automata as an alternative representation of sets of sequences, and a linear translation of future LTL formulas into alternating automata. Section 3 describes the three algorithms for checking traces for correctness, and in Section 4 these algorithms are extended to collect statistics from traces related to the desired temporal property. Section 5 extends the trace-checking algorithm to be applicable to formulas with both future and past temporal operators. The implementation of the trace-checking algorithms and the results of some experimental runs are presented in Section 6.

2. Preliminaries

2.1. Specification Language: Linear-Time Temporal Logic

The specification language we use in this paper is *linear-time temporal logic*. A *temporal formula* is constructed out of state formulas, which can be either propositional or first-order formulas, to which we apply the boolean connectives and the temporal operators shown below.

Traditionally, temporal formulas are interpreted over a *model*, which is an infinite sequence of states $\sigma : s_0, s_1, \ldots$. Given a model σ, a state formula p and temporal formulas φ and ψ, we present an inductive definition for the notion of a formula φ holding at a position $j \geq 0$ in σ, denoted by $(\sigma, j) \models \varphi$ (Manna and Pnueli, 1995).

For a state formula:

$$(\sigma, j) \models p \quad \text{iff} \quad s_j \models p, \text{ that is, } p \text{ holds on state } s_j.$$

For the boolean connectives:

$$\begin{array}{lll}
(\sigma, j) \models \varphi \wedge \psi & \text{iff} & (\sigma, j) \models \varphi \text{ and } (\sigma, j) \models \psi \\
(\sigma, j) \models \varphi \vee \psi & \text{iff} & (\sigma, j) \models \varphi \text{ or } (\sigma, j) \models \psi \\
(\sigma, j) \models \neg\varphi & \text{iff} & (\sigma, j) \not\models \varphi \ .
\end{array}$$

For the future temporal operators:

$$\begin{array}{lll}
(\sigma, j) \models \bigcirc \varphi & \text{iff} & (\sigma, j+1) \models \varphi \\
(\sigma, j) \models \square \varphi & \text{iff} & (\sigma, i) \models \varphi \text{ for all } i \geq j \\
(\sigma, j) \models \diamondsuit \varphi & \text{iff} & (\sigma, i) \models \varphi \text{ for some } i \geq j \\
(\sigma, j) \models \varphi \, \mathcal{U} \, \psi & \text{iff} & (\sigma, k) \models \psi \text{ for some } k \geq j, \\
 & & \quad \text{and } (\sigma, i) \models \varphi \text{ for every } i,\ j \leq i < k \\
(\sigma, j) \models \varphi \, \mathcal{W} \, \psi & \text{iff} & (\sigma, j) \models \varphi \, \mathcal{U} \, \psi \text{ or } (\sigma, j) \models \square \varphi \ .
\end{array}$$

For the past temporal operators:

$$\begin{array}{lll}
(\sigma, j) \models \ominus \varphi & \text{iff} & j > 0 \text{ and } (\sigma, j-1) \models \varphi \\
(\sigma, j) \models \widetilde{\ominus} \varphi & \text{iff} & j = 0 \text{ or } (\sigma, j-1) \models \varphi \\
(\sigma, j) \models \boxminus \varphi & \text{iff} & (\sigma, i) \models \varphi \text{ for all } 0 \leq i \leq j \\
(\sigma, j) \models \diamondminus \varphi & \text{iff} & (\sigma, i) \models \varphi \text{ for some } 0 \leq i \leq j \\
(\sigma, j) \models \varphi \, \mathcal{S} \, \psi & \text{iff} & (\sigma, k) \models \psi \text{ for some } k \leq j, \\
 & & \quad \text{and } (\sigma, i) \models \varphi \text{ for every } i,\ k < i \leq j \\
(\sigma, j) \models \varphi \, \mathcal{B} \, \psi & \text{iff} & (\sigma, j) \models \varphi \, \mathcal{S} \, \psi \text{ or } (\sigma, j) \models \boxminus \varphi \ .
\end{array}$$

An infinite sequence of states σ *satisfies* a temporal formula φ, written $\sigma \models \varphi$, if $(\sigma, 0) \models \varphi$. The set of all sequences that satisfy a formula φ is denoted by $\mathcal{L}(\varphi)$, the *language* of φ.

We say that a formula is a future (past) formula if it contains only state formulas, boolean connectives and future (past) temporal operators.

2.1.1. *Finite Models*

In runtime verification temporal formulas must generally be interpreted over finite sequences of states. For a finite sequence of states, $\sigma : s_0, s_1, \ldots, s_{n-1}$ we adapt the definitions of the semantics of the future temporal operators as follows:

$$\begin{array}{lll}
(\sigma, j) \models \bigcirc \varphi & \text{iff} & j < n-1 \text{ and } (\sigma, j+1) \models \varphi \\
(\sigma, j) \models \square \varphi & \text{iff} & (\sigma, i) \models \varphi \text{ for all } i,\ j \leq i < n \\
(\sigma, j) \models \diamondsuit \varphi & \text{iff} & (\sigma, i) \models \varphi \text{ for some } i,\ j \leq i < n \\
(\sigma, j) \models \varphi \, \mathcal{U} \, \psi & \text{iff} & (\sigma, k) \models \psi \text{ for some } j \leq k < n, \\
 & & \quad \text{and } (\sigma, i) \models \varphi \text{ for every } i,\ j \leq i < k \\
(\sigma, j) \models \varphi \, \mathcal{W} \, \psi & \text{iff} & (\sigma, j) \models \varphi \, \mathcal{U} \, \psi \text{ or } (\sigma, j) \models \square \varphi \ .
\end{array}$$

It follows from the definition that $\bigcirc \varphi$ is always false at the last state of the trace, and $\diamondsuit \varphi$ and $\psi \, \mathcal{U} \, \varphi$ are true at the last state of a sequence only if φ is true at that state.

The definitions of the past temporal operators remain unchanged for finite sequences of states.

2.2. Alternating Automata

Automata on infinite words, also known as ω-automata (Thomas, 1990), are a convenient way to represent temporal formulas. For every linear temporal formula there exists an automaton on infinite words such that a sequence of states satisfies the temporal formula if and only if it is accepted by the corresponding automaton. Thus to check whether a trace satisfies a particular temporal formula, we can check whether it is accepted by the corresponding automaton.

Alternating automata (Vardi, 1996) are a generalization of nondeterministic automata and $\forall$-automata (Manna and Pnueli, 1987). Nondeterministic automata have an existential flavor: a word is accepted if it is accepted by *some* path through the automaton. $\forall$-automata, on the other hand have a universal flavor: a word is accepted if it is accepted by *all* paths through the automaton. Alternating automata combine the two flavors by allowing choices along a path to be marked as either existential or universal.

The advantage of using alternating automata to represent the language accepted by a temporal formula is that the alternating automaton that accepts the same language as the formula is linear in the size of the formula, while it is worst-case exponential for nondeterministic or $\forall$-automata.

We describe a translation from temporal formulas to alternating automata on finite words, applying the interpretation of temporal formulas over finite sequences defined in Section 2.1.1. The corresponding translation for the traditional interpretation over infinite sequences can be found in (Manna and Sipma, 2000).

Definition 1 (Alternating Automaton) An *alternating automaton* $\mathcal{A}$ is defined as follows:

$\mathcal{A}$ ::=	$\epsilon_{\mathcal{A}}$	empty automaton
|	$\langle \nu, \mathcal{A}, f \rangle$	single node
|	$\mathcal{A} \wedge \mathcal{A}$	conjunction of two automata
|	$\mathcal{A} \vee \mathcal{A}$	disjunction of two automata

where ν is a state formula and f is a boolean indicating whether the node it is part of is accepting, indicated by *acc*, or rejecting, indicated by *rej*. We require that the automaton be finite.

In this data structure a node $n : \langle \nu, \delta, f \rangle$, can be viewed as an automaton state, with labeling $\nu(n)$, next-state relation $\delta(n)$ and indication of inclusion in the acceptance set $f(n)$.

The set of nodes of an alternating automaton $\mathcal{A}$, denoted by $\mathcal{N}(\mathcal{A})$ is defined as:

$$
\begin{array}{lcl}
\mathcal{N}(\epsilon_{\mathcal{A}}) & = & \emptyset \\
\mathcal{N}(\langle \nu, \delta, f \rangle) & = & \{\langle \nu, \delta, f \rangle\} \cup \mathcal{N}(\delta) \\
\mathcal{N}(\mathcal{A}_1 \wedge \mathcal{A}_2) & = & \mathcal{N}(\mathcal{A}_1) \cup \mathcal{N}(\mathcal{A}_2) \\
\mathcal{N}(\mathcal{A}_1 \vee \mathcal{A}_2) & = & \mathcal{N}(\mathcal{A}_1) \cup \mathcal{N}(\mathcal{A}_2)
\end{array}
$$

A path through a nondeterministic automaton is a sequence of nodes. A "path" through an alternating ω-automaton is, in general, a tree.

Definition 2 A *tree* is defined recursively as follows:

$$
\begin{array}{rcll}
T & ::= & \epsilon_T & \text{empty tree} \\
& | & T \cdot T & \text{composition} \\
& | & \langle \langle \nu, \delta, f \rangle, T \rangle & \text{single node with child tree}
\end{array}
$$

Definition 3 (Run) Given a finite sequence of states $\sigma : s_0, \ldots, s_{n-1}$ and an automaton $\mathcal{A}$, a tree T is called a *run* of σ in $\mathcal{A}$ if one of the following holds:

$$
\begin{array}{lcl}
\mathcal{A} = \epsilon_{\mathcal{A}} & \text{and} & T = \epsilon_T \\
\mathcal{A} = \langle \nu, \delta, f \rangle & \text{and} & n > 1,\ T = \langle \langle \nu, \delta, f \rangle, T' \rangle,\ s_0 \vDash \nu \text{ and} \\
& & T' \text{ is a run of } s_1, \ldots, s_{n-1} \text{ in } \delta, \text{ or} \\
& & n = 1,\ T = \langle \langle \nu, \delta, f \rangle, \epsilon_T \rangle \text{ and } s_0 \vDash \nu \\
\mathcal{A} = \mathcal{A}_1 \wedge \mathcal{A}_2 & \text{and} & T = T_1 \cdot T_2, \\
& & T_1 \text{ is a run of } \sigma \text{ in } \mathcal{A}_1 \text{ and} \\
& & T_2 \text{ is a run of } \sigma \text{ in } \mathcal{A}_2 \\
\mathcal{A} = \mathcal{A}_1 \vee \mathcal{A}_2 & \text{and} & T_1 \text{ is a run of } \sigma \text{ in } \mathcal{A}_1 \text{ or} \\
& & T_2 \text{ is a run of } \sigma \text{ in } \mathcal{A}_2
\end{array}
$$

Definition 4 (Accepting run) A run is *accepting* if every path through the tree ends in an accepting node.

Definition 5 (Model) A finite sequence of states σ is a *model* of an alternating automaton $\mathcal{A}$ if there exists an accepting run of σ in $\mathcal{A}$.

The set of models of an automaton $\mathcal{A}$, also called the *language of* $\mathcal{A}$, is denoted by $\mathcal{L}(\mathcal{A})$.

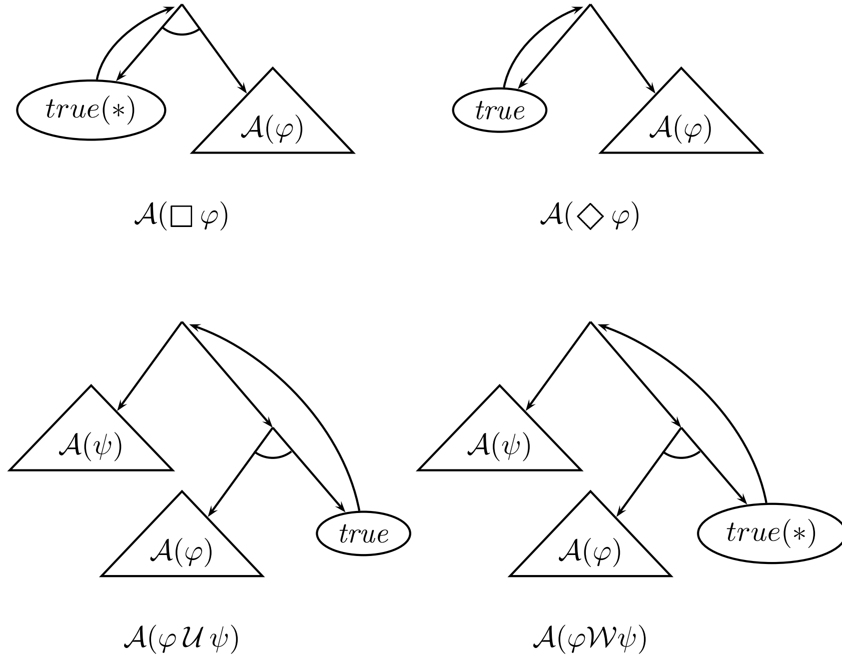

Figure 1. Alternating automata for the temporal operators $\Box$, $\Diamond$, $\mathcal{U}$, $\mathcal{W}$

2.3. Linear-Time Temporal Logic: Future Formulas

It has been shown that for every LTL formula φ there exists an alternating automaton $\mathcal{A}$ such that $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A})$ and the size of $\mathcal{A}$ is linear in the size of φ (Vardi, 1997). The construction of the automaton follows the structure of the formula, where we assume that all negations have been pushed in to the state level (a full set of rewrite rules to accomplish this is given in (Manna and Pnueli, 1995)), that is, no temporal operator is in the scope of a negation. Given an LTL formula φ, an alternating automaton $\mathcal{A}(\varphi)$ is constructed as follows:
For a state formula p:

$$\mathcal{A}(p) = \langle p, \epsilon_{\mathcal{A}}, acc \rangle \ .$$

For temporal formulas φ and ψ:

$$\begin{array}{lcl}
\mathcal{A}(\varphi \wedge \psi) & = & \mathcal{A}(\varphi) \wedge \mathcal{A}(\psi) \\
\mathcal{A}(\varphi \vee \psi) & = & \mathcal{A}(\varphi) \vee \mathcal{A}(\psi) \\
\mathcal{A}(\bigcirc \varphi) & = & \langle true, \mathcal{A}(\varphi), rej \rangle \\
\mathcal{A}(\Box \varphi) & = & \langle true, \mathcal{A}(\Box \varphi), acc \rangle \wedge \mathcal{A}(\varphi) \\
\mathcal{A}(\Diamond \varphi) & = & \langle true, \mathcal{A}(\Diamond \varphi), rej \rangle \vee \mathcal{A}(\varphi) \\
\mathcal{A}(\varphi \, \mathcal{U} \, \psi) & = & \mathcal{A}(\psi) \vee (\langle true, \mathcal{A}(\varphi \, \mathcal{U} \, \psi), rej \rangle \wedge \mathcal{A}(\varphi)) \\
\mathcal{A}(\varphi \mathcal{W} \psi) & = & \mathcal{A}(\psi) \vee (\langle true, \mathcal{A}(\varphi \mathcal{W} \psi), acc \rangle \wedge \mathcal{A}(\varphi))
\end{array}$$

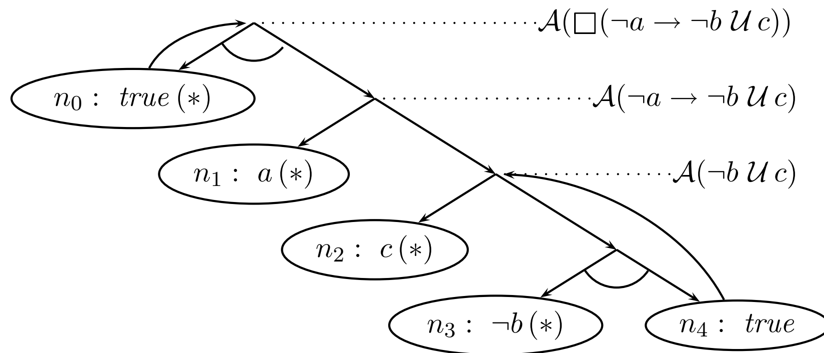

Figure 2. Alternating automaton for $\Box(\neg a \to \neg b\,\mathcal{U}\,c)$

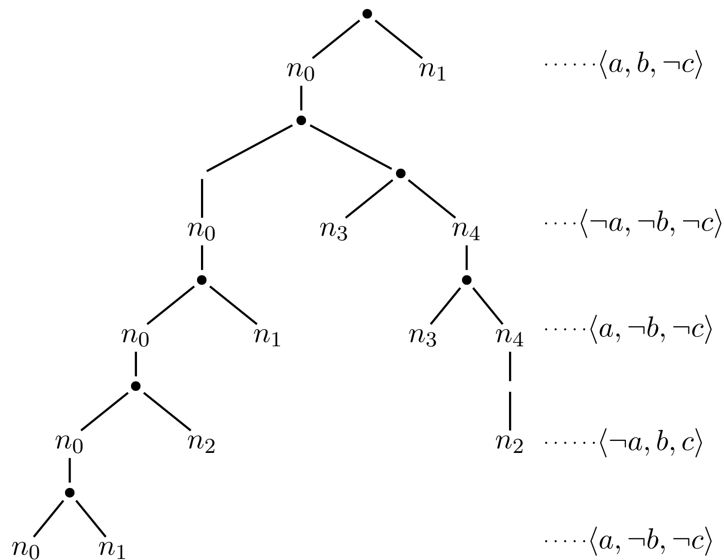

Figure 3. Run in the automaton for $\Box(\neg a \to \neg b\,\mathcal{U}\,c)$

The constructions are illustrated in Figure 1, where accepting nodes are marked by an asterisk.

Example
Figure 2 shows the automaton for the formula $\Box(\neg a \to \neg b\,\mathcal{U}\,c)$, which states that from every position at which a is false, b should be false until c becomes true. A model for this formula, for example, is the trace

$$\langle a, b, \neg c\rangle, \langle \neg a, \neg b, \neg c\rangle, \langle a, \neg b, \neg c\rangle, \langle \neg a, b, c\rangle, \langle a, \neg b, \neg c\rangle$$

where the tuples show the truth assignments for the three propositional variables a, b, and c at successive positions in the trace. An accepting run for this trace is shown in Figure 3.

3. Checking Traces

We present three algorithms to check whether a finite sequence of states, also referred to as a *trace*, satisfies a temporal formula, using the finite semantics of temporal formulas introduced in the previous section. All are based on alternating automata, but make different optimizations and are favored in different situations.

Given a trace σ and a specification φ, the first algorithm attempts to construct an accepting run, starting from the root of the tree, of σ in $\mathcal{A}(\varphi)$, by recursively traversing the automaton in a depth-first manner. The second algorithm attempts to construct an accepting run of σ in $\mathcal{A}(\varphi)$, by traversing the automaton in a breadth-first manner: for each position in the trace it creates all possible combinations of nodes that can be part of a run for that trace at that position. The third algorithm traverses the trace backwards while attempting to construct an accepting run starting from the leaves of the tree. In this case no search has to be performed: the value of each node in the tree can be computed by simply looking up the values of its child nodes.

In the three algorithms we assume that a trace consists of a finite sequence of states and that we have some way of checking whether a state satisfies a propositional or first-order formula.

3.1. Depth-first traversal

The depth-first traversal algorithm is the easiest to implement. To check whether a trace σ satisfies a temporal formula φ, we first generate the alternating automaton $\mathcal{A}(\varphi)$ for φ and then recursively check for the existence of an accepting run, following the structure of the automaton.

To prepare for the depth-first algorithm we restate Definition 5 as a recursive function. Given an automaton $\mathcal{A}$, a trace σ and a position n, the value of $\textsc{ct}(\mathcal{A}, \sigma, n)$ is *true* iff there is a run of σ in $\mathcal{A}$, starting with position n.

For $n < |\sigma| - 1$:

$$\begin{array}{lcl} \textsc{ct}(\mathcal{A}_1 \wedge \mathcal{A}_2, \sigma, n) & = & \textsc{ct}(\mathcal{A}_1, \sigma, n) \wedge \textsc{ct}(\mathcal{A}_2, \sigma, n) \\ \textsc{ct}(\mathcal{A}_1 \vee \mathcal{A}_2, \sigma, n) & = & \textsc{ct}(\mathcal{A}_1, \sigma, n) \vee \textsc{ct}(\mathcal{A}_2, \sigma, n) \\ \textsc{ct}(\langle \nu, \delta, f \rangle, \sigma, n) & = & \sigma[n] \models \nu \ \wedge \textsc{ct}(\delta, \sigma, n+1) \end{array}$$

```
DEPTH-FIRST(A, σ, n)
 1: if A = ε_A then
 2:    return true
 3: else if A = A_1 ∧ A_2 then
 4:    return DEPTH-FIRST(A_1, σ, n) ∧ DEPTH-FIRST(A_2, σ, n)
 5: else if A = A_1 ∨ A_2 then
 6:    return DEPTH-FIRST(A_1, σ, n) ∨ DEPTH-FIRST(A_2, σ, n)
 7: else if A = ⟨ν, δ, f⟩ then
 8:    if σ[n] ⊨ ν then
 9:       if n = |σ| − 1 then
10:          return (f = acc)
11:       else
12:          return DEPTH-FIRST(δ, σ, n + 1)
13:       end if
14:    else
15:       return false
16:    end if
17: end if
```

Figure 4. Depth-first traversal algorithm.

For the last position of the trace, $n = |\sigma| - 1$, we apply Definition 4:

$$\begin{array}{rcl} \textsc{ct}(\langle \nu, \delta, acc \rangle, \sigma, n) & = & \sigma[n] \models \nu \\ \textsc{ct}(\langle \nu, \delta, rej \rangle, \sigma, n) & = & \mathit{false} \end{array}$$

Figure 4 shows the resulting depth-first algorithm that recursively computes the value of $\textsc{ct}(\mathcal{A}, \sigma, n)$. The function call $\textsc{Depth-first}(\mathcal{A}, \sigma, 0)$ returns *true* iff σ is a model of $\mathcal{A}$.

Clearly this algorithm is potentially inefficient in that it may traverse parts of the trace multiple times due to the recursive calls for conjunction and disjunction. As an example, consider the specification

$$\Box \Diamond \varphi.$$

Suppose φ is satisfied only at the last element of the trace. At each position the algorithm will traverse the remainder of the trace to look for φ. In the worst case, there is a recursive call for *every node* in the automaton. Therefore this algorithm becomes prohibitively slow for long traces.

3.2. Breadth-first traversal

An alternative approach is to search for an accepting run in a breadth-first manner. We construct the run tree $\mathcal{T}$ as a sequence of slices $C_i \subseteq V$, where the slice C_i contains a node n iff there is a path of length i from the root of the tree to n. The breadth-first algorithm traverses the trace only once, by keeping track of the set of slices that are consistent with the trace prefix seen so far[1].

Figure 5 shows the breadth-first traversal algorithm. The function call Breadth-first($\mathcal{A}, \sigma$) returns *true* iff σ is a model of $\mathcal{A}$. The variable S stores the set of slices that are consistent with the trace prefix seen so far. Initially, this set is computed with the function *initial*($\mathcal{A}$). This function computes the set of all possible slices that can appear as slice C_0 of some run in $\mathcal{A}$. (Note that function *initial*($\mathcal{A}$) does not depend on the trace.)

$$\begin{aligned}
\mathit{initial}(\epsilon_{\mathcal{A}}) &= \{\emptyset\} \\
\mathit{initial}(\langle \nu, \delta, f \rangle) &= \{\{\langle \nu, \delta, f \rangle\}\} \\
\mathit{initial}(\mathcal{A}_1 \wedge \mathcal{A}_2) &= \mathit{initial}(\mathcal{A}_1) \otimes \mathit{initial}(\mathcal{A}_2) \\
\mathit{initial}(\mathcal{A}_1 \vee \mathcal{A}_2) &= \mathit{initial}(\mathcal{A}_1) \cup \mathit{initial}(\mathcal{A}_2)
\end{aligned}$$

where $\otimes$ denotes the crossproduct:

$$\{S_1, \ldots, S_n\} \otimes \{T_1, \ldots, T_m\} = \{S_i \cup T_j \mid i = 1 \ldots n, j = 1 \ldots m\}$$

Example
We apply the function *initial* to the automaton for $\Box(\neg a \rightarrow \neg b \,\mathcal{U}\, c)$, shown in Figure 2:

$$\mathit{initial}(\mathcal{A}(\Box(\neg a \rightarrow \neg b \,\mathcal{U}\, c))) = \{\{n_0, n_1\}, \{n_0, n_2\}, \{n_0, n_3, n_4\}\}$$

$\Box$

In the body of the main loop (lines 3–10) the set of slices S is updated as follows: First, function *state-satisfied*(C, s) checks, for a given slice C from the set, and a trace state s, if C is satisfied in s:

$$\textit{state-satisfied}(C, s) = \text{for all nodes } \langle \nu, \delta, f \rangle \in C : \; s \models \nu.$$

[1] Note that this construction will necessarily produce identical subtrees whenever the same automaton node is reached on different paths of the same length. This can be done without loss of generality, since, given an accepting run in which the subtrees are different, we can always construct another accepting run in which they are identical, simply by replacing one of the subtrees with the other.

BREADTH-FIRST($\mathcal{A}, \sigma$)
```
1: S ← initial(A)
2: for n = 0 to |σ| − 2 do
3:    S' ← ∅
4:    for each C ∈ S do
5:       if state-satisfied(C, σ[n]) then
6:          S' ← S' ∪ successors(C)
7:       end if
8:    end for
9:    S ← S'
10: end for
11: S' ← ∅
12: for each C ∈ S do
13:    if state-satisfied(C, σ[|σ| − 1]) and accepting(C) then
14:       S' ← S' ∪ C
15:    end if
16: end for
17: return (S' ≠ ∅)
```

Figure 5. Breadth-first traversal algorithm.

For those slices that are state-satisfied, function *successors*(C) computes the set of candidate successor slices:

$$successors(C) = \bigotimes_{n \in C} initial(\delta(n)).$$

When the traversal reaches the last position of the trace (lines 12–17), function *accepting*(C) is used to check if a slice contains only accepting nodes: the trace is accepted if some slice in S is state-satisfied in the last position and contains only accepting nodes.

$$accepting(C) = \text{for all nodes } \langle \nu, \delta, f \rangle \in C: \;\; f = acc.$$

Example
We illustrate the algorithm on the automaton for $\Box(\neg a \rightarrow \neg b \, \mathcal{U} \, c)$ and the trace

$$\langle a, b, \neg c \rangle, \langle \neg a, \neg b, \neg c \rangle, \langle a, \neg b, \neg c \rangle, \langle \neg a, b, c \rangle, \langle a, \neg b, \neg c \rangle$$

presented before in Section 2.3. Starting with the set of slices

$$\{\{n_0, n_1\}, \{n_0, n_2\}, \{n_0, n_3, n_4\}\}$$

as computed by function *initial*, we detect that only the slice $\{n_0, n_1\}$ is state-satisfied for the first state of the trace, $\langle a, b, \neg c\rangle$. Computing the successors of $\{n_0, n_1\}$ we again obtain

$$\{\{n_0, n_1\}, \{n_0, n_2\}, \{n_0, n_3, n_4\}\}$$

as candidate slices for the second slice. For the second state in the trace, $\langle \neg a, \neg b, \neg c\rangle$, only the slice $\{n_0, n_3, n_4\}$ is state-satisfied. To compute the successors of this slice, we take the crossproduct of the successors of n_0,n_3, and n_4, that is

$$\{\{n_0, n_1\}, \{n_0, n_2\}, \{n_0, n_3, n_4\}\} \otimes \{\emptyset\} \otimes \{\{n_2\}, \{n_3, n_4\}\}$$

resulting in

$$\{\{n_0, n_1, n_2\}, \{n_0, n_1, n_3, n_4\}, \{n_0, n_2\}, \{n_0, n_2, n_3, n_4\}, \{n_0, n_3, n_4\}\}$$

of which both the slices $\{n_0, n_1, n_3, n_4\}$ and $\{n_0, n_3, n_4\}$ are state-satisfied for the third state, $\langle a, \neg b, \neg c\rangle$. Again we get the same set of successors, but for the fourth state, $\langle \neg a, b, c\rangle$, only the slice $\{n_0, n_2\}$ is state-satisfied. Finally, for the fifth slice we compute candidates

$$\{\{n_0, n_1\}, \{n_0, n_2\}, \{n_0, n_3, n_4\}\}$$

where $\{n_0, n_1\}$ is the only slice that is both accepting and state-satisfied. Having found, upon completion of the traversal, a nonempty set of slices, the algorithm returns *true*, as expected.

The two runs observed during the execution of the algorithm are thus

$$\begin{array}{cc} \{n_0, n_1\} & \{n_0, n_1\} \\ \{n_0, n_3, n_4\} & \{n_0, n_3, n_4\} \\ \{n_0, n_1, n_3, n_4\} & \{n_0, n_3, n_4\} \\ \{n_0, n_2\} & \{n_0, n_2\} \\ \{n_0, n_1\} & \{n_0, n_1\} \end{array}$$

the first of which is the same as the run shown in Figure 3. □

The breadth-first algorithm clearly traverses the trace only once. However, it may generate an exponential number of sets of nodes at each position in the trace. We have found that for typical formulas the number of sets is relatively small, however for larger formulas this may be a problem as is illustrated by one of the examples presented in Section 6.

3.3. Reverse Traversal

The blow-up in the number of sets in the breadth-first traversal is caused by nondeterminism in the formula. This can be avoided, as was pointed out by Roşu and Havelund, by traversing the trace backwards (Roşu and Havelund, 2001; Havelund and Rosu, 2002b).

Like the depth-first algorithm, the reverse traversal algorithm computes the value of the function $\textsc{ct}(\mathcal{A}, \sigma, n)$ at position $n = 0$ of the trace. The difference with the depth-first algorithm is that we now refer to previously computed values rather than making a recursive call. Figure 6 shows the reverse traversal algorithm. We use an array *result* to store the previously computed values. The function $\textsc{Reverse}(\mathcal{A}, \sigma)$ initializes *result* at the last position of the trace (where only accepting nodes are considered) and then successively updates the array for earlier positions, using the auxiliary function *eval* to compute the result for disjunctions, conjunctions, and the empty automaton, as follows:

$$\begin{aligned}
&eval(\epsilon_{\mathcal{A}}, result) &&= true \\
&eval(\langle \nu, \delta, f \rangle, result) &&= result[\langle \nu, \delta, f \rangle] \\
&eval(\mathcal{A}_1 \wedge \mathcal{A}_2, result) &&= eval(\mathcal{A}_1, result) \wedge eval(\mathcal{A}_2, result) \\
&eval(\mathcal{A}_1 \vee \mathcal{A}_2, result) &&= eval(\mathcal{A}_1, result) \vee eval(\mathcal{A}_2, result)
\end{aligned}$$

Example
Consider again the trace and automaton presented in Section 2.3. The successive values computed for *result* are, starting from the last state in the trace:

	n_0	n_1	n_2	n_3	n_4
$4 : \langle a, \neg b, \neg c \rangle$:	*true*	*true*	*false*	*true*	*false*
$3 : \langle \neg a, b, c \rangle$:	*true*	*false*	*true*	*false*	*false*
$2 : \langle a, \neg b, \neg c \rangle$:	*true*	*true*	*false*	*true*	*true*
$1 : \langle \neg a, \neg b, \neg c \rangle$:	*true*	*false*	*false*	*true*	*true*
$0 : \langle a, b, \neg c \rangle$:	*true*	*true*	*false*	*false*	*false*

After having computed the values in *result* for the first position in the trace, the algorithm computes

$$eval(result, \mathcal{A}_{\Box(\neg a \rightarrow \neg b\, \mathcal{U}\, c)})$$

which evaluates to

$$result[n_0] \wedge (result[n_1] \vee result[n_2] \vee (result[n_3] \wedge result[n_4]))$$

and thus, with the above values, evaluates to *true*, indicating that a run exists. □

```
REVERSE(A, σ)
 1: for each ⟨ν, δ, f⟩ ∈ N(A) do
 2:    if (f = acc) then
 3:       result[⟨ν, δ, f⟩] ← (σ[|σ| − 1] ⊨ ν)
 4:    else
 5:       result[⟨ν, δ, f⟩] ← false
 6:    end if
 7: end for
 8: for n = |σ| − 1  downto 0 do
 9:    for each ⟨ν, δ, f⟩ ∈ N(A) do
10:       if σ[n] ⊨ ν then
11:          result′[⟨ν, δ, f⟩] ← eval(result, δ)
12:       else
13:          result′[⟨ν, δ, f⟩] ← false
14:       end if
15:    end for
16:    result ← result′
17: end for
18: return eval(result, A)
```

Figure 6. Reverse traversal algorithm.

4. Collecting Statistics over Traces

The trace checking algorithms considered so far have been concerned with the *correctness* of a system behavior: does the trace *satisfy* the specification? Since a trace is only a finite prefix of the infinite system behavior, this question is not always meaningful. The liveness property $\Box \Diamond p$ ("infinitely often p"), for example, is satisfied in any trace in which p is *true* in the last state — even if this is the only state in which p is *true*. On the other hand, $\Box \Diamond p$ is not satisfied on a trace in which p is *false* in the last state, even if p is *true* in all other states.

Collecting statistics over traces is a way to get a better impression to what extent eventualities are fulfilled. For example, we can count how often a run visits rejecting nodes, and report the minimum number over all accepting runs. A low number indicates that the eventualities have been fulfilled quickly.

Another interesting statistic is *specification coverage:* which nodes are visited by all accepting runs? If large parts of the automaton are avoided, this may indicate a problem: perhaps parts of the specification are vacuously true, or the trace is too short to exhibit all aspects of the specified behavior.

In this section we extend the trace checking algorithms to collect statistics during their traversal of the automaton. We assume here that the statistic of interest can be defined directly in terms of the automaton. A further extension of this approach is to work with a temporal logic that combines the temporal specification with the collection of statistical data. We have started to explore this approach in a separate paper (Finkbeiner et al., 2002).

4.1. Representing Statistics

Trace checking computes a boolean value: *true* if the trace has an accepting run, *false* otherwise. We now generalize the analysis to return statistical data. We restrict ourselves to data that can be represented as a lattice $\mathcal{S}$ with a meet operation $\underline{\wedge}$, a join operation $\underline{\vee}$, a bottom element $\perp$, and a top element $\top$. Under this assumption, the collection of statistics becomes a straightforward extension of trace checking, where conjunctions in the alternating automaton cause meets on the statistical data, and disjunctions in the automaton cause joins on the statistical data[2]. In addition to the lattice operations, we assume there is a special operation $update(s, n)$ that returns, for given statistical data s and an automaton node n, the new statistical data after the node n has been traversed. Trace checking is a special case, with $\underline{\wedge} = \wedge, \underline{\vee} = \vee$, $\perp = false$, $\top = true$ and $update(S, n) = S$.

Example
To illustrate the mechanism for collecting statistics, we present a simple example in which we count the number of visits to rejecting nodes and report the minimum number of all accepting runs. $\perp$ means that there are no accepting runs; $\top$ means that there is an accepting run without visits to rejecting nodes, i.e., $\top = 0$. A conjunction corresponds to a branching of the run tree, we therefore implement $\underline{\wedge}$ as addition; Disjunctions allow for multiple runs, we therefore implement $\underline{\vee}$ as minimum. When visiting a rejecting node, the *update* function adds 1; when

[2] Our approach is inspired by Bruns and Godefroid's *extended alternating automata,* introduced for the purpose of temporal logic query checking (Bruns and Godefroid, 2001).

visiting an accepting node the value is left unchanged.

$$
\begin{array}{lcl}
\mathcal{S} & = & \mathbb{N} \cup \{\bot\} \\
\top & = & 0 \\
s_1 \mathbin{\underline{\wedge}} s_2 & = & \begin{cases} \bot & \text{if } s_1 = \bot \text{ or } s_2 = \bot; \\ s_1 + s_2 & \text{otherwise.} \end{cases} \\
s_1 \mathbin{\underline{\vee}} s_2 & = & \begin{cases} s_1 & \text{if } s_2 = \bot; \\ s_2 & \text{if } s_1 = \bot; \\ \min\{s_1, s_2\} & \text{otherwise.} \end{cases} \\
\mathit{update}(s, \langle \nu, \delta, f \rangle) & = & \begin{cases} s & \text{if } f = \mathit{acc} \\ \bot & \text{if } s = \bot \\ s + 1 & \text{otherwise.} \end{cases}
\end{array}
$$

Suppose we evaluate the automaton for $\neg a \rightarrow \neg b \ \mathcal{U} \ c$, shown as a subautomaton of the automaton presented in Figure 2, over the trace

$$\langle a, \neg b, \neg c \rangle, \langle a, \neg b, c \rangle$$

There are two accepting runs: One that only visits n_1, and a second run that first visits nodes n_3 and n_4 and then n_2. The first run does not visit any rejecting nodes. Hence, the result is 0. The trace

$$\langle \neg a, \neg b, \neg c \rangle, \langle \neg a, \neg b, c \rangle$$

on the other hand, allows only the second run. The result for this trace is therefore 1. □

Example

As a second example, we evaluate the same automaton and the same traces as in the first example, but compute a different statistic: we determine which nodes are visited by all accepting runs. $\bot$ again means that there are no accepting runs; $\top$ means that no nodes are visited, i.e., $\top = \emptyset$; a non-empty subset of $\mathcal{N}(\mathcal{A})$ means that the indicated nodes are visited by all accepting runs. A conjunction combines two subtrees; correspondingly, we implement $\underline{\wedge}$ as set union. Since we are interested in the nodes that are visited by *all* accepting runs, $\underline{\vee}$ is implemented as set intersection. The *update* function simply adds the presently visited

node to the set.

$$
\begin{array}{lcl}
\mathcal{S} & = & 2^{\mathcal{N}(\mathcal{A})} \cup \{\bot\} \\
\top & = & \emptyset \\
s_1 \mathrel{\underline{\wedge}} s_2 & = & \begin{cases} \bot & \text{if } s_1 = \bot \text{ or } s_2 = \bot; \\ s_1 \cup s_2 & \text{otherwise.} \end{cases} \\
s_1 \mathrel{\underline{\vee}} s_2 & = & \begin{cases} s_1 & \text{if } s_2 = \bot; \\ s_2 & \text{if } s_1 = \bot; \\ s_1 \cap s_2 & \text{otherwise.} \end{cases} \\
\mathit{update}(s, n) & = & \begin{cases} \bot & \text{if } s = \bot; \\ s \cup \{n\} & \text{otherwise.} \end{cases}
\end{array}
$$

Consider again the automaton for $\neg a \rightarrow \neg b \,\mathcal{U}\, c$, shown in Figure 2, and the trace

$$\langle a, \neg b, \neg c\rangle, \langle a, \neg b, c\rangle$$

The two accepting runs visit disjoint sets of nodes (node n_1, and nodes n_2, n_3, n_4, respectively). The result, therefore, is the empty set. The trace

$$\langle \neg a, \neg b, \neg c\rangle, \langle \neg a, \neg b, c\rangle$$

on the other hand, forces a visit to n_2, n_3 and n_4. □

4.2. Depth-first and reverse traversal

In a depth-first or reverse traversal, statistics can be collected by evaluating the following function STAT instead of the function CT used for trace checking. For positions $n < |\sigma| - 1$:

$$
\begin{array}{lcl}
\mathrm{STAT}(\epsilon_{\mathcal{A}}, \sigma, n) & = & \top \\
\mathrm{STAT}(\mathcal{A}_1 \wedge \mathcal{A}_2, \sigma, n) & = & \mathrm{STAT}(\mathcal{A}_1, \sigma, n) \mathrel{\underline{\wedge}} \mathrm{STAT}(\mathcal{A}_2, \sigma, n) \\
\mathrm{STAT}(\mathcal{A}_1 \vee \mathcal{A}_2, \sigma, n) & = & \mathrm{STAT}(\mathcal{A}_1, \sigma, n) \mathrel{\underline{\vee}} \mathrm{STAT}(\mathcal{A}_2, \sigma, n) \\
\mathrm{STAT}(\langle \nu, \delta, f\rangle, \sigma, n) & = & \begin{cases} \mathit{update}(\mathrm{STAT}(\delta, \sigma, n+1), \langle \nu, \delta, f\rangle) & \text{if } \sigma[n] \vDash \nu \\ \bot & \text{otherwise.} \end{cases}
\end{array}
$$

At the end of the trace, that is, at position $n = |\sigma| - 1$, only accepting nodes can have a value different from $\bot$:

$$
\begin{array}{lcl}
\mathrm{STAT}(\langle \nu, \delta, \mathit{acc}\rangle, \sigma, n) & = & \begin{cases} \mathit{update}(\top, \langle \nu, \delta, \mathit{acc}\rangle) & \text{if } \sigma[n] \vDash \nu \\ \bot & \text{otherwise.} \end{cases} \\
\mathrm{STAT}(\langle \nu, \delta, \mathit{rej}\rangle, \sigma, n) & = & \bot
\end{array}
$$

Example
We apply the depth-first algorithm to compute the statistic from the first example (the minimum number of visits to rejecting nodes). Consider again the automaton for $\neg a \rightarrow \neg b \,\mathcal{U}\, c$ (Figure 2) and the trace

$$\sigma : \langle a, \neg b, \neg c \rangle, \langle a, \neg b, c \rangle.$$

Using the depth-first algorithm, we evaluate

$$\textsc{stat}(\mathcal{A}_{\neg a \rightarrow \neg b \,\mathcal{U}\, c}, \sigma, 0) = \begin{pmatrix} \textsc{stat}(n_1, \sigma, 0) \veebar \textsc{stat}(n_2, \sigma, 0) \\ \veebar \\ \textsc{stat}(n_3, \sigma, 0) \,\underline{\wedge}\, \textsc{stat}(n_4, \sigma, 0) \end{pmatrix}.$$

Nodes n_1, n_2 and n_3 have no successors. Furthermore, the state at the first position, $\langle a, \neg b, \neg c \rangle$, satisfies the labels of n_1 and n_3, but not the label of n_2. For nodes n_1, n_2 and n_3, the recursive evaluation therefore terminates with

$$\begin{aligned} \textsc{stat}(n_1, \sigma, 0) &= \mathit{update}(\top, n_1) = 0, \\ \textsc{stat}(n_2, \sigma, 0) &= \bot, \\ \textsc{stat}(n_3, \sigma, 0) &= \mathit{update}(\top, n_3) = 0. \end{aligned}$$

Node n_4 has the successor automaton $\mathcal{A}_{\neg b \,\mathcal{U}\, c}$, thus

$$\textsc{stat}(n_4, \sigma, 0) = \mathit{update}(\textsc{stat}(\mathcal{A}_{\neg b \,\mathcal{U}\, c}, \sigma, 1), n_4).$$

At position 1 we obtain

$$\textsc{stat}(\mathcal{A}_{\neg b \,\mathcal{U}\, c}, \sigma, 1) = \begin{pmatrix} \textsc{stat}(n_2, \sigma, 1) \\ \veebar \\ \textsc{stat}(n_3, \sigma, 1) \,\underline{\wedge}\, \textsc{stat}(n_4, \sigma, 1) \end{pmatrix},$$

where

$$\begin{aligned} \textsc{stat}(n_2, \sigma, 1) &= \mathit{update}(\top, n_2) = 0 \\ \textsc{stat}(n_3, \sigma, 1) &= \mathit{update}(\top, n_3) = 0, \\ \textsc{stat}(n_4, \sigma, 1) &= \bot. \end{aligned}$$

Hence,

$$\begin{aligned} \textsc{stat}(\mathcal{A}_{\neg b \,\mathcal{U}\, c}, \sigma, 1) &= 0, \\ \textsc{stat}(n_4, \sigma, 0) &= \mathit{update}(0, n_4) = 1. \end{aligned}$$

From this we conclude at position 0:

$$\textsc{stat}(\mathcal{A}_{\neg a \rightarrow \neg b \,\mathcal{U}\, c}, \sigma, 1) = \min\{0, 1\} = 0.$$

□

4.3. Breadth-first traversal

The gathering of statistical information in a depth-first or reverse traversal can be seen as an annotation of each node in a run with a statistical summary of the possible subtrees starting in that node. When statistics are collected in a breadth-first traversal, we annotate each slice with statistical information that summarizes the run up to the current slice.

The breadth-first trace checking algorithm was presented in Section 3.2. We now describe changes to the algorithm that replace each slice C with an annotated slice $\langle C, s\rangle$ where s is the statistical information.

Function *initial* (which computes the set of slices that appear as slice C_0 of some run) is extended as follows:

$$\begin{array}{lcl} initial(\epsilon_{\mathcal{A}}) & = & \{\langle\emptyset, \bot\rangle\} \\ initial(\langle\nu,\delta,f\rangle) & = & \{\langle\{\langle\nu,\delta,f\rangle\}, update(\top, \langle\nu,\delta,f\rangle)\rangle\} \\ initial(\mathcal{A}_1 \wedge \mathcal{A}_2) & = & initial(\mathcal{A}_1) \otimes initial(\mathcal{A}_2) \\ initial(\mathcal{A}_1 \vee \mathcal{A}_2) & = & initial(\mathcal{A}_1) \cup initial(\mathcal{A}_2) \end{array}$$

where $\otimes$ denotes the crossproduct:

$$\{\langle C_1, s_1\rangle, \ldots, \langle C_n, s_n\rangle\} \otimes \{\langle C'_1, s'_1\rangle, \ldots, \langle C'_m, s'_m\rangle\} = \left\{\langle C_i \cup C'_j, s_i \bigtriangleup s'_j\rangle \mid i = 1 \ldots n, j = 1 \ldots m\right\}$$

As described in Section 3.2, the breadth-first algorithm maintains a set of slices, that is initially computed by function *initial*. For each position in the trace, the *successors* function determines the successors of the state-satisfied slices in that set. The *successors* function is extended as follows:

$$successors\ (\langle C, s\rangle) = \bigotimes_{n\in C} \kappa(\delta(n), s)$$

where

$$\begin{array}{lcl} \kappa(\epsilon_{\mathcal{A}}, s) & = & \{\langle\emptyset, s\rangle\} \\ \kappa(\langle\nu,\delta,f\rangle, s) & = & \{\langle\{\langle\nu,\delta,f\rangle\}, update(s, \langle\nu,\delta,f\rangle)\rangle\} \\ \kappa(\mathcal{A}_1 \wedge \mathcal{A}_2, s) & = & \kappa(\mathcal{A}_1, s) \otimes \kappa(\mathcal{A}_2, s) \\ \kappa(\mathcal{A}_1 \vee \mathcal{A}_2, s) & = & \kappa(\mathcal{A}_1, s) \cup \kappa(\mathcal{A}_2, s) \end{array}$$

At the last position, slices that are either not state-satisfied or that contain rejecting nodes are eliminated. The algorithm then returns the join of all remaining annotations.

Example
Again we compute the minimum number of visits to rejecting nodes), using the automaton for $\neg a \rightarrow \neg b \, \mathcal{U} \, c$ (Figure 2) and the trace

$$\sigma : \langle a, \neg b, \neg c \rangle, \langle a, \neg b, c \rangle$$

Initially, we generate the following set of annotated slices:

$$\mathit{initial}(\mathcal{A}_{\neg a \rightarrow \neg b \, \mathcal{U} \, c}) = \{(\{n_1\}, 0), (\{n_2\}, 0), (\{n_3, n_4\}, 1)\}$$

In position 0 of the trace only

$$(\{n_1\}, 0) \text{ and } (\{n_3, n_4\}, 1)$$

are state-satisfied. The successors are

$$\{(\emptyset, 0), (\{n_3, n_4\}, 2), (\{n_2\}, 1)\},$$

of which $(\emptyset, 0)$ and $(\{n_2\}, 1)$ remain after $(\{n_3, n_4\}, 2)$ is eliminated because of the rejecting node n_4. Both slices are state-satisfied. The result is again

$$0 \veebar 1 = \min\{0, 1\} = 0.$$

□

5. Past Temporal Operators

The algorithms presented in the previous sections are applicable to LTL formulas with future temporal operators only. It is relatively straightforward to generalize the algorithms to include past temporal operators as well. In this section we give an outline of the necessary extensions.

5.1. Alternating Automata with Past Nodes

To define an alternating automaton for LTL formulas including past operators, we add a component g to the definition of a node, such that a node is defined as

$$\langle \nu, \delta, f, g \rangle$$

where g indicates whether the node is a *past node* (indicated by "$\leftarrow$"), a *future node* (indicated by "$\rightarrow$"), or a *state node* (indicated by "$\downarrow$"). The definition of a run of such an alternating automaton reflects the presence of past nodes as follows:

Definition 6 (Run) Given a finite sequence of states $\sigma : s_0, s_1, \ldots, s_n$, an automaton $\mathcal{A}$, and a position $j \geq 0$, a tree T is called a run of σ in $\mathcal{A}$ at position j if one of the following holds:

$\mathcal{A} = \epsilon_{\mathcal{A}}$	and	$T = \epsilon_T$
$\mathcal{A} = \langle \nu, \delta, f, \downarrow \rangle$	and	$T = \langle \langle \nu, \delta, f, \downarrow \rangle, \epsilon_T \rangle$ and $s_j \vDash \nu$;
$\mathcal{A} = \langle \nu, \delta, f, \rightarrow \rangle$	and	$T = \langle \langle \nu, \delta, f, \rightarrow \rangle, T' \rangle$ and $s_j \vDash \nu$ and T' is a run of σ in δ at $j+1$;
$\mathcal{A} = \langle \nu, \delta, f, \leftarrow \rangle$	and	$T = \langle \langle \nu, \delta, f, \leftarrow \rangle, T' \rangle$ and $s_j \vDash \nu$ and $\begin{cases} \text{(a)} & 0 < j < n-1 \text{ and } T' \text{ is a run of } \sigma \text{ in } \delta \text{ at position } j-1, \text{ or} \\ \text{(b)} & j = 0 \text{ and } T' = \epsilon_T, \text{ or} \\ \text{(c)} & j = n-1 \text{ and } T' = \epsilon_T; \end{cases}$
$\mathcal{A} = \mathcal{A}_1 \wedge \mathcal{A}_2$	and	$T = T_1 \cdot T_2$, T_1 is a run of $\mathcal{A}_1$ at position j and T_2 is a run of $\mathcal{A}_2$ at position j;
$\mathcal{A} = \mathcal{A}_1 \vee \mathcal{A}_2$	and	T_1 is a run of $\mathcal{A}_1$ at position j or T_2 is a run of $\mathcal{A}_2$ at position j

The definitions for accepting run and model are the same as before.

Definition 7 (Accepting run) A run is *accepting* if every path through the tree ends in an accepting node.

Note that the end of a path in the tree may now be a past or state node evaluated at the first position in the trace.

Definition 8 (Model) A finite sequence of states σ is a *model* of an alternating automaton $\mathcal{A}$ if there exists an accepting run of σ in $\mathcal{A}$.

5.2. Linear Temporal Logic

Given an LTL formula φ, an alternating automaton $\mathcal{A}(\varphi)$ is constructed as before, with

$$\mathcal{A}(p) = \langle p, \epsilon_{\mathcal{A}}, acc, \downarrow \rangle$$

for a state formula, and constructions for the future operators identical to those presented in Section 2.3 except for the addition of the $\rightarrow$ component to each node. For temporal formulas φ and ψ the constructions for the past formulas are as follows:

$$\begin{array}{lcl}
\mathcal{A}(\circleddash\!\!\!\sim \varphi) & = & \langle \mathit{true}, \mathcal{A}(\varphi), \mathit{acc}, \leftarrow \rangle \\
\mathcal{A}(\ominus \varphi) & = & \langle \mathit{true}, \mathcal{A}(\varphi), \mathit{rej}, \leftarrow \rangle \\
\mathcal{A}(\boxminus \varphi) & = & \langle \mathit{true}, \mathcal{A}(\boxminus \varphi), \mathit{acc}, \leftarrow \rangle \wedge \mathcal{A}(\varphi) \\
\mathcal{A}(\diamondminus \varphi) & = & \langle \mathit{true}, \mathcal{A}(\diamondminus \varphi), \mathit{rej}, \leftarrow \rangle \vee \mathcal{A}(\varphi) \\
\mathcal{A}(\varphi \, \mathcal{S} \, \psi) & = & \mathcal{A}(\psi) \vee (\langle \mathit{true}, \mathcal{A}(\varphi \, \mathcal{S} \, \psi), \mathit{rej}, \leftarrow \rangle \wedge \mathcal{A}(\varphi)) \\
\mathcal{A}(\varphi \, \mathcal{B} \, \psi) & = & \mathcal{A}(\psi) \vee (\langle \mathit{true}, \mathcal{A}(\varphi \, \mathcal{B} \, \psi), \mathit{acc}, \leftarrow \rangle \wedge \mathcal{A}(\varphi))
\end{array}$$

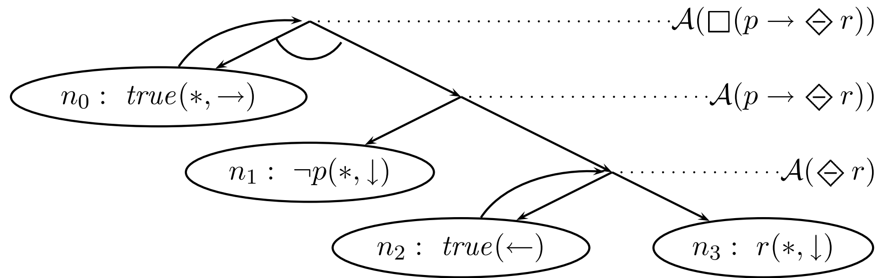

Figure 7. Alternating automaton for $\square(p \to \diamondminus r)$

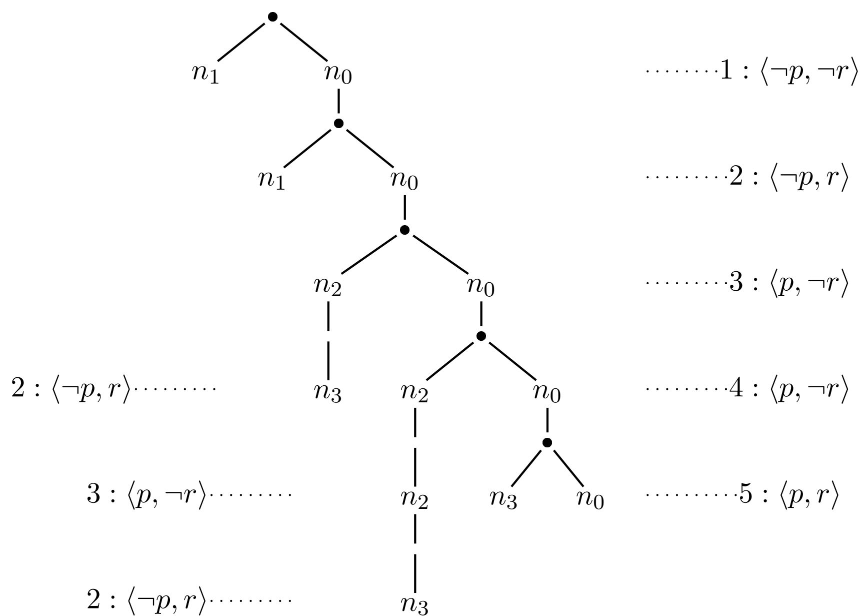

Figure 8. Run in the automaton for $\varphi : \square(p \to \diamondminus r)$

Example
Figure 7 shows the automaton for the formula $\varphi : \square(p \to \diamondminus r)$ with p and r state formulas. This formula states that every occurrence of p is preceded by an occurrence of r, and hence is called a *causality* formula

(Manna and Pnueli, 1995). As before, accepting nodes are marked by an asterisk.

Figure 8 shows a run for the trace

$$\sigma : \langle \neg p, \neg r \rangle, \langle \neg p, r \rangle, \langle p, \neg r \rangle, \langle p, \neg r \rangle, \langle p, r \rangle$$

in the the automaton for φ. Notice that slices of the tree may now contain nodes evaluated over different positions in the trace.

□

5.3. Depth-first algorithm

The depth-first algorithm presented in Section 3.1 can be extended to include past operators by adding the following cases:

$$\begin{array}{lll} \textsc{ct}(\langle \nu, \delta, f, \downarrow \rangle, \sigma, n) & = \sigma[n] \models \nu & \\ \textsc{ct}(\langle \nu, \delta, f, \leftarrow \rangle, \sigma, n) & = \sigma[n] \models \nu \ \wedge \textsc{ct}(\delta, \sigma, n-1) & n > 0 \\ \textsc{ct}(\langle \nu, \delta, acc, \leftarrow \rangle, \sigma, 0) & = \sigma[0] \models \nu & \\ \textsc{ct}(\langle \nu, \delta, rej, \leftarrow \rangle, \sigma, 0) & = \mathit{false} & \end{array}$$

The other cases remain unchanged.

5.4. Breadth-first algorithm

Adapting the breadth-first algorithm to be applicable to formulas containing past operators is more complex. In the presence of past operators, nodes in a slice of a run may refer to different positions of the trace, and therefore extra bookkeeping is required.

To simplify the algorithm we restrict ourselves here to formulas in which no future operator occurs in the scope of a past operator. That is, the direction of time is reversed at most once in a run. This restriction allows us to apply the reverse-traversal algorithm of section 3.3 to that part of the run in which the direction of time is towards the past, and use the result to prune the successors of each slice in the forward direction.

To prepare for the algorithm, we first identify the nodes in the automaton that can participate in the reverse-traversal algorithm, that is, all nodes whose evaluation does not depend on the evaluation of future nodes. We call these nodes the backward looking nodes of an automaton, denoted by $\mathcal{N}_B(\mathcal{A})$. It is easy to see that in the automaton for a formula with the above restriction, all nodes reachable from a past node are either past nodes or state nodes. For example, in Figure 7 only nodes n_2 and n_3 are reachable from past node n_2. Therefore all past nodes and their successors are backward looking nodes. More formally,

given an automaton $\mathcal{A}$ we denote with $\mathcal{N}_B(\mathcal{A}) \subseteq \mathcal{N}(\mathcal{A})$ the backward looking nodes of $\mathcal{A}$, defined as

$$\begin{array}{lcl}
\mathcal{N}_B(\epsilon_{\mathcal{A}}) & = & \emptyset \\
\mathcal{N}_B(\langle \nu, \delta, f, \downarrow \rangle) & = & \emptyset \\
\mathcal{N}_B(\langle \nu, \delta, f, \rightarrow \rangle) & = & \mathcal{N}_B(\delta) \\
\mathcal{N}_B(\langle \nu, \delta, f, \leftarrow \rangle) & = & \{\langle \nu, \delta, f, \leftarrow \rangle\} \cup \mathcal{N}(\delta) \\
\mathcal{N}_B(\mathcal{A}_1 \wedge \mathcal{A}_2) & = & \mathcal{N}_B(\mathcal{A}_1) \cup \mathcal{N}_B(\mathcal{A}_2) \\
\mathcal{N}_B(\mathcal{A}_1 \vee \mathcal{A}_2) & = & \mathcal{N}_B(\mathcal{A}_1) \cup \mathcal{N}_B(\mathcal{A}_2)
\end{array}$$

Note that in the fourth line we conjoin the past node with all nodes of the next-state automaton.

We now describe the algorithm Breadth-first-past, shown in Figure 5.4, which checks for the existence of an accepting run of a trace σ in an automaton $\mathcal{A}$ in a breadth-first fashion. Lines 1–7 initialize the values for the backward looking nodes in the array *past*, similar to the first part of algorithm Reverse in Section 3.3. In line 8 we initialize the first set of candidate slices for the first position of the run, similar to the first line of algorithm Breadth-first in Section 3.2. However, here we use the values of the backward looking nodes to eliminate slices that are known to be unsatisfiable. To achieve this, we redefine the function *initial* as follows

$$\begin{array}{lcl}
\mathit{initial}(\epsilon_{\mathcal{A}}, \mathit{past}) & = & \{\emptyset\} \\
\mathit{initial}(\langle \nu, \delta, f, \downarrow \rangle, \mathit{past}) & = & \{\{\langle \nu, \delta, f, \downarrow \rangle\}\} \\
\mathit{initial}(\langle \nu, \delta, f, \rightarrow \rangle, \mathit{past}) & = & \{\{\langle \nu, \delta, f, \rightarrow \rangle\}\} \\
\mathit{initial}(\langle \nu, \delta, f, \leftarrow \rangle, \mathit{past}) & = & \begin{cases} \emptyset \text{ if } \mathit{past}[\langle \nu, \delta, f, \leftarrow \rangle] = \mathit{false} \\ \{\langle \nu, \delta, f, \leftarrow \rangle\} \text{ otherwise} \end{cases} \\
\mathit{initial}(\mathcal{A}_1 \wedge \mathcal{A}_2, \mathit{past}) & = & \mathit{initial}(\mathcal{A}_1, \mathit{past}) \otimes \mathit{initial}(\mathcal{A}_2, \mathit{past}) \\
\mathit{initial}(\mathcal{A}_1 \vee \mathcal{A}_2, \mathit{past}) & = & \mathit{initial}(\mathcal{A}_1, \mathit{past}) \cup \mathit{initial}(\mathcal{A}_2, \mathit{past})
\end{array}$$

where, as in Section 3.2, $\otimes$ denotes the crossproduct:

$$\{S_1, \ldots, S_n\} \otimes \{T_1, \ldots, T_m\} = \{S_i \cup T_j \mid i = 1 \ldots n, j = 1 \ldots m\}$$

Lines 9–25 proceed to traverse the trace as in Breadth-first with the difference that at the start of the computation of each new set of candidate slices, we first compute the values of the backward looking nodes for the next position, based on their values in the current position (lines 10–17) as in the Reverse algorithm, where the function *eval* is defined similarly as in Section 3.3:

$$\begin{array}{lcl}
\mathit{eval}(\epsilon_{\mathcal{A}}, \mathit{past}) & = & \mathit{true} \\
\mathit{eval}(\langle \nu, \delta, f, g \rangle, \mathit{past}) & = & \mathit{past}[\langle \nu, \delta, f, g \rangle] \\
\mathit{eval}(\mathcal{A}_1 \wedge \mathcal{A}_2, \mathit{past}) & = & \mathit{eval}(\mathcal{A}_1, \mathit{past}) \wedge \mathit{eval}(\mathcal{A}_2, \mathit{past}) \\
\mathit{eval}(\mathcal{A}_1 \vee \mathcal{A}_2, \mathit{past}) & = & \mathit{eval}(\mathcal{A}_1, \mathit{past}) \wedge \mathit{eval}(\mathcal{A}_2, \mathit{past})
\end{array}$$

```
BREADTH-FIRST-PAST(𝒜, σ)
 1: for each ⟨ν, δ, f, g⟩ ∈ 𝒩_B(𝒜) do
 2:    if (f = acc) then
 3:       past[⟨ν, δ, f, g⟩] ← (σ[0] ⊨ ν)
 4:    else
 5:       past[⟨ν, δ, f, g⟩] ← false
 6:    end if
 7: end for
 8: S ← initial(𝒜, past)
 9: for n = 0 to |σ| − 2 do
10:    for each ⟨ν, δ, f, g⟩ ∈ 𝒩_B(𝒜) do
11:       if σ[n + 1] ⊨ ν then
12:          past′[⟨ν, δ, f, g⟩] ← eval(past, δ)
13:       else
14:          past′[⟨ν, δ, f, g⟩] ← false
15:       end if
16:    end for
17:    past ← past′
18:    S′ ← ∅
19:    for each C ∈ S do
20:       if state-satisfied(C, σ[n]) then
21:          S′ ← S ∪ successors(C, past)
22:       end if
23:    end for
24:    S ← S′
25: end for
26: S′ ← ∅
27: for each C ∈ S do
28:    if state-satisfied(C, σ[|σ| − 1]) and accepting(C) then
29:       S′ ← S ∪ C
30:    end if
31: end for
32: return (S′ ≠ ∅)
```

Figure 9. Breadth-first algorithm for automata with past nodes.

Lines 19–23 then compute the set of candidate slices, again using the values of the backward looking nodes to eliminate slices whose past nodes are not satisfied. The function *successors* is redefined as

$$successors(C, past) = \bigotimes_{n \in C} initial(\delta(n), past)$$

Finally, lines 27–31 evaluate the set of slices at the last position and keep only those slices that are state-satisfied and accepting. The algorithm returns *true* if the remaining set of slices is nonempty.

Example
To illustrate the algorithm we apply it to the automaton $\mathcal{A}$ for the formula $\Box(p \rightarrow \diamondminus r)$, shown in Figure 7, and the trace

$$\sigma : \langle\neg p, \neg r\rangle, \langle\neg p, r\rangle, \langle p, \neg r\rangle, \langle p, \neg r\rangle, \langle p, r\rangle$$

for which a run was shown in Figure 8.

First, the array *past* is initialized on the first state of the trace. $past[n_2] = false$ because n_2 is rejecting, $past[n_3] = false$ as $\langle\neg p, \neg r\rangle \not\models r$.

Using these values for *past*, S is initialized with the set of candidate slices for the first position:

$$S \ = \ \{\{n_0\}\} \otimes (\{\{n_1\}\} \cup \emptyset \cup \emptyset) = \{\{n_0, n_1\}\}$$

For the second slice new values are computed for *past*. Now $past[n_2] = false$ because $past[n_2] \vee past[n_3] = false$. Slice $\{\{n_0, n_1\}\}$ is found to be state-satisfied and thus its successors are computed using the new values of *past*, resulting in

$$S = \{\{n_0\}\} \otimes (\{\{n_1\}\} \cup \emptyset \cup \{\{n_3\}\}) = \{\{n_0, n_1\}, \{n_0, n_3\}\}$$

of which only $\{n_0, n_3\}$ is state-satisfied. In the two subsequent passes through the loop S is set to

$$\{\{n_0, n_1\}, \{n_0, n_2\}\}$$

as now $past[n_2]$ is *true* and will remain *true*. For $n = 2$ and $n = 3$ both sets are state-satisfied. For the last position $past[n_3] = true$ and thus the candidate set is

$$\{\{n_0, n_1\}, \{n_0, n_2\}, \{n_0, n_3\}\}$$

of which all are state-satisfied. The slices $\{n_0, n_1\}$ and $\{n_0, n_3\}$ are accepting, and therefore the algorithm returns *true*. □

6. Implementation and Experiments

The algorithms were implemented in Java, making use of existing software modules for expression parsing, propositional simplification and generation of alternating automata available in the STeP (Stanford

Temporal Prover) system (Bjørner et al., 2000). The size of the programs implementing the three trace checking algorithms are 75, 190, and 80 lines of code respectively. No attempts were made to optimize the code except for caching of successor sets in the breadth-first algorithm (which resulted in a speed-up of a factor 5) and caching of results in the reverse traversal algorithms.

The three algorithms were applied to the following three temporal formulas:

$$\varphi_1 : \Box \Diamond z$$
$$\varphi_2 : \Box \Diamond a$$
$$\varphi_3 : \Box(b \rightarrow \neg a \, \mathcal{U} \, (a \, \mathcal{U} \, (\neg a \, \mathcal{U} \, a)))$$

For all formulas traces were generated randomly containing states in the following proportions:

$$\begin{array}{ll} \langle a, \neg b, \neg c, \neg z \rangle & : 10\% \\ \langle \neg a, b, \neg c, \neg z \rangle & : 40\% \\ \langle \neg a, \neg b, c, \neg z \rangle & : 25\% \\ \langle \neg a, \neg b, \neg c, \neg z \rangle & : 25\% \end{array}$$

For φ_1 a single state $\langle \neg a, \neg b, \neg c, z \rangle$ was added to the end of the trace and for φ_2 and φ_3 a $\langle a, \neg b, \neg c, \neg z \rangle$-state was added to the end to ensure satisfaction for easier comparison. The running times are presented in Figure 10. The program was run on a 1.7GHz PC, running Redhat Linux v7.0 and Sun JDK1.3.1.

The results confirm our expectations. Indeed the depth-first algorithm performs poorly on the formula $\Box \Diamond z$, while the two other algorithms can deal with this case easily. When eventualities are fulfilled reasonably quickly, as is the case with $\Box \Diamond a$ (as roughly in every tenth trace element a is true), the performance of the depth-first algorithm is comparable with the other two. For large formulas, such as $\Box(b \rightarrow \neg a \, \mathcal{U} \, (a \, \mathcal{U} \, (\neg a \, \mathcal{U} \, a)))$, the breadth-first algorithm performs considerably worse than the other two, due to the large number of sets to maintain at each position in the trace. Again here eventualities are fulfilled relatively quickly and therefore the performance of the depth-first algorithm is comparable to that of the reverse traversal.

Based on these preliminary results, it is clear that all three algorithms have their utility. Reverse traversal is always the preferred choice if it is possible. However, in many situations, especially online monitoring, this is not an option. In that case depth-first checking is feasible if waiting times are not too long (and there are no disjunctions in eventualities), especially if one wants to gather statistics on these waiting times. For long waiting times or in the presence of disjunctions on eventualities, and relatively small formulas breadth-first is preferred.

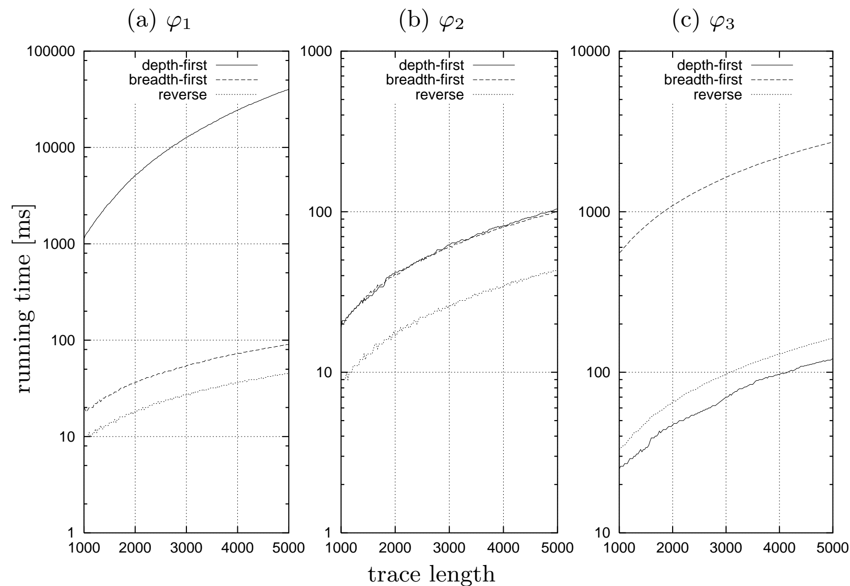

Figure 10. Running times for trace checking.

Acknowledgements

We thank Klaus Havelund and Grigore Roşu for bringing the topic of checking finite traces to our attention at the NASA/RIACS workshop on Validation and Verification, December 2000, and Grigore for his discussion on various approaches. We also thank the anonymous referees for their numerous comments and suggestions.

References

Bjørner, N. S., A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H. B. Sipma, and T. E. Uribe: 2000, 'Verifying Temporal Properties of Reactive Systems: A STeP Tutorial'. *Formal Methods in System Design* **16**(3), 227–270.

Bruns, G. and P. Godefroid: 2001, 'Temporal Logic Query Checking'. In: *Proc. 16th IEEE Symp. Logic in Comp. Sci.* pp. 409–417, IEEE Computer Society Press.

Drusinsky, D.: 2000, 'The Temporal Rover and the ATG Rover'. In: K. Havelund, J. Penix, and W. Visser (eds.): *SPIN Model Checking and Software Verification, 7th Int'l SPIN Workshop*, Vol. 1885 of *LNCS*. pp. 323–330, Springer-Verlag.

Finkbeiner, B., S. Sankaranarayanan, and H. B. Sipma: 2002, 'Collecting Statistics over Runtime Executions'. In: K. Havelund and G. Rosu (eds.): *Runtime Verification 2002*, Vol. 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier.

Havelund, K.: 2000, 'Using Runtime Analysis to Guide Model Checking of Java Programs'. In: K. Havelund, J. Penix, and W. Visser (eds.): *SPIN Model Checking*

and Software Verification, 7th Int'l SPIN Workshop, Vol. 1885 of *LNCS*. pp. 245–264, Springer-Verlag.

Havelund, K. and G. Roşu: 2001, 'Testing Linear Temporal Logic Formulae on Finite Execution Traces'. Technical Report TR 01-08, RIACS.

Havelund, K. and G. Rosu (eds.): 2001, 'Runtime Verification 2001', Vol. 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers.

Havelund, K. and G. Rosu (eds.): 2002a, 'Runtime Verification 2001', Vol. 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers.

Havelund, K. and G. Rosu: 2002b, 'Synthesizing Monitors for Safety Properties'. In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, Vol. 2280 of *LNCS*. pp. 342–356, Springer-Verlag.

Kim, M., S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan: 2001, 'Java-MaC: a Run-time Assurance Tool for Java Programs'. In: K. Havelund and G. Rosu (eds.): *Runtime Verification (RV 2001)*, Vol. 55 of *Electronic Notes in Computer Science*. Paris, pp. 115–132, Elsevier Science Publishers.

Lee, I., S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan: 1999, 'Runtime Assurance Based on Formal Specifications'. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*.

Manna, Z. and A. Pnueli: 1987, 'Specification and Verification of Concurrent Programs by ∀-automata'. In: B. Banieqbal, H. Barringer, and A. Pnueli (eds.): *Temporal Logic in Specification*, No. 398 in LNCS. Berlin: Springer-Verlag, pp. 124–164. Also in *Proc. 14th ACM Symp. Princ. of Prog. Lang.*, Munich, Germany, pp. 1–12, January 1987.

Manna, Z. and A. Pnueli: 1995, *Temporal Verification of Reactive Systems: Safety*. New York: Springer-Verlag.

Manna, Z. and H. B. Sipma: 2000, 'Alternating the Temporal Picture for Safety'. In: U. Montanari, J. D. Rolim, and E. Welzl (eds.): *Proc. 27th Intl. Colloq. Aut. Lang. Prog.*, Vol. 1853. Geneva, Switzerland, pp. 429–450, Springer-Verlag.

Roşu, G. and K. Havelund: 2001, 'Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae'. Technical Report TR 01-15, RIACS.

Thomas, W.: 1990, 'Automata on infinite objects'. In: J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science*, Vol. B. Elsevier Science Publishers (North-Holland), pp. 133–191.

Vardi, M. Y.: 1995, 'Alternating Automata and Program Verification'. In: J. van Leeuwen (ed.): *Computer Science Today. Recent Trends and Developments*, Vol. 1000 of *LNCS*. Springer-Verlag, pp. 471–485.

Vardi, M. Y.: 1996, 'An Automata-Theoretic Approach to Linear Temporal Logic'. In: F. Moller and G. Birtwistle (eds.): *Logics for Concurrency. Structure versus Automata*, Vol. 1043 of *LNCS*. pp. 238–266, Springer-Verlag.

Vardi, M. Y.: 1997, 'Alternating Automata: Checking Truth and Validity for Temporal Logics'. In: *Proc. 14th Intl. Conference on Automated Deduction*, Vol. 1249 of *LNCS*. Springer-Verlag.