Checking Finite Traces using Alternating Automata

Bernd Finkbeiner and Henny Sipma

Computer Science Department, Stanford University
Stanford, CA. 94305

Abstract

We present three algorithms to check at runtime whether a reactive program satisfies a temporal specification, expressed by a future linear-time temporal logic formula. The three methods are all based on alternating automata, but traverse the automaton in different ways: depth-first, breadth-first, and backwards, respectively. All three methods have been implemented and experimental results are presented. We outline an extension to these algorithms that is applicable to LTL formulas containing both past and future operators.

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. For safety properties one can generate the automaton for the property and check the trace against the automaton until a trace state is found that is not implied by the corresponding automaton state, in which case a violation is found, or until the end of the trace is reached, in which case the trace satisfies the property. Liveness properties, of course, can never be falsified on a finite trace. However, one may wish to get an impression to what extent eventualities are fulfilled in a finite trace.

In this paper we present three algorithms to check whether a finite trace satisfies a temporal specification. All three algorithms are based on alternating automata, but traverse the trace in different ways, appropriate for different situations. Having three algorithms based on the same automaton allows us to easily vary the runtime characteristics without changing the semantics.

[1] 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.

Checking of finite traces has received a growing attention recently. Examples include the commercial system Temporal Rover [Dru00], 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 [Hav00]. The work presented in this paper was inspired by [HR01,RH01].

The rest 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: depth-first traversal, breadth-first traversal, and backwards traversal. In Section 4 we propose a method 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 Temporal Logic

The specification language we use in this paper is *linear 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.

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 $(\sigma, j) \models \varphi$.
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:

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

For the future temporal operators:

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

For the past temporal operators:

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

An infinite sequence of states σ *satisfies* a temporal formula ϕ, written $\sigma \models \phi$, if $(\sigma, 0) \models \phi$. 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. We say that a formula is a general safety formula if it is of the form $\square \varphi$, for a past formula φ.

2.2 Alternating Automata

Automata on infinite words [Tho90] 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 alternating automaton.

Several types of automata on infinite words exist. Alternating automata [Var96] are a generalization of nondeterministic automata and $\forall$-automata [MP87]. 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. Below we will show a one-to-one mapping from the temporal formula to the automaton.

We briefly summarize our definition of alternating automata. A more elaborate description of our version of alternating automata can be found in [MS00].

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

$$\begin{array}{rlll} \mathcal{A} ::= & \epsilon_{\mathcal{A}} & & \text{empty automaton} \\ \mid & \langle \nu, \delta, f \rangle & & \text{single node} \\ \mid & \mathcal{A} \wedge \mathcal{A} & & \text{conjunction of two automata} \\ \mid & \mathcal{A} \vee \mathcal{A} & & \text{disjunction of two automata} \end{array}$$

where ν is a state formula, δ is an alternating automaton expressing the next-state relation, and f indicates whether the node is accepting (denoted by *acc*), rejecting (denoted by *rej*), or final (denoted by *fin*). We require that the automaton be finite.

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

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

A path through a regular ω-automaton is an infinite sequence of nodes. A "path" through an alternating ω-automaton is, in general, a directed acyclic graph (dag).

Definition 2 (Run) Given an infinite sequence of states $\sigma : s_0, s_1, \ldots$, a labeled dag $\langle V_0, V, E, \mu \rangle$ with nodes V, root nodes $V_0 \subseteq V$, edge function $E : V \to 2^V$, and a node labeling $\mu : V \to \mathcal{N}(\mathcal{A})$ is called a *run* of σ in $\mathcal{A}$ if one of the following holds:

$$\begin{array}{lll}
\mathcal{A} = \epsilon_{\mathcal{A}} & \text{and} & V_0 = \emptyset \\
\mathcal{A} = n & \text{and} & s_0 \vDash \nu(n) \text{ and} \\
 & & \text{there is a node } m \in V_0 \text{ s.t. } \mu(m) = n \text{ and} \\
 & & \langle E(m), V, E \rangle \text{ is a run of } s_1, s_2, \ldots \text{ in } \delta(n) \\
\mathcal{A} = \mathcal{A}_1 \wedge \mathcal{A}_2 & \text{and} & \langle V_0, V, E \rangle \text{ is a run in } \mathcal{A}_1 \text{ and} \\
 & & \langle V_0, V, E \rangle \text{ is a run in } \mathcal{A}_2 \\
\mathcal{A} = \mathcal{A}_1 \vee \mathcal{A}_2 & \text{and} & \langle V_0, V, E \rangle \text{ is a run in } \mathcal{A}_1 \text{ or} \\
 & & \langle V_0, V, E \rangle \text{ is a run in } \mathcal{A}_2
\end{array}$$

Definition 3 (Accepting run) A run is *accepting* if every infinite path contains infinitely many accepting nodes.

Definition 4 (Model) An infinite 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 $\mathcal{L}(\mathcal{A})$.

2.3 Linear 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 [Var97]. In our construction of the automaton 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 [MP95]), 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 [MS00].
For a state formula p:

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

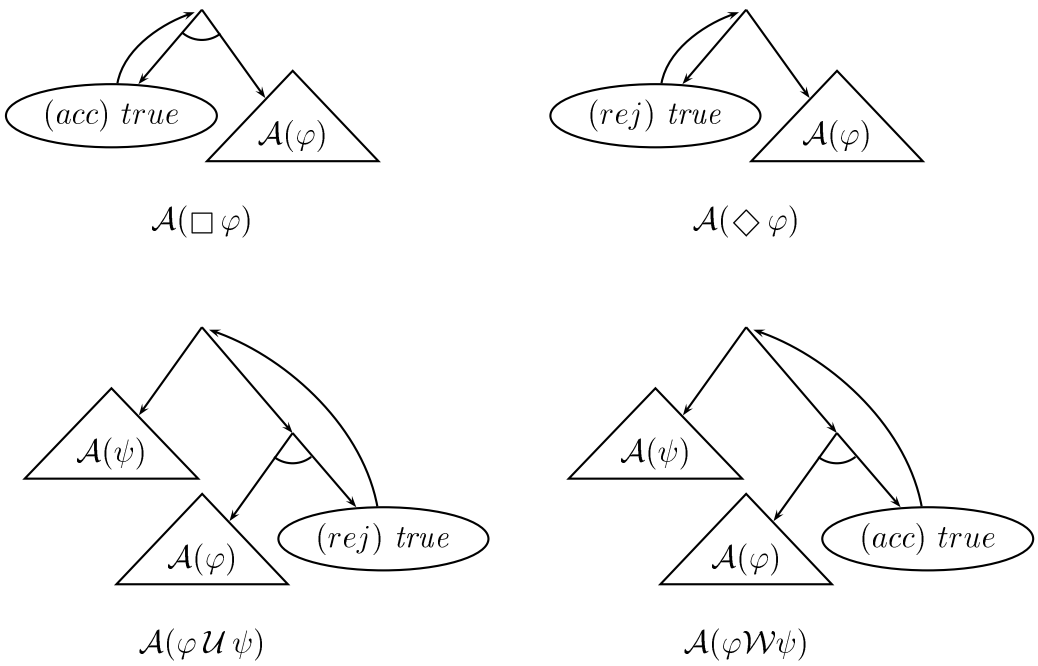

Fig. 1. Alternating automata for the temporal operators $\square$, $\diamond$, $\mathcal{U}$, $\mathcal{W}$

For temporal formulas φ and ψ:

$$
\begin{aligned}
\mathcal{A}(\varphi \wedge \psi) &= \mathcal{A}(\varphi) \wedge \mathcal{A}(\psi) \\
\mathcal{A}(\varphi \vee \psi) &= \mathcal{A}(\varphi) \vee \mathcal{A}(\psi) \\
\mathcal{A}(\bigcirc \varphi) &= \langle \mathit{true}, \mathcal{A}(\varphi), \mathit{rej} \rangle \\
\mathcal{A}(\square \varphi) &= \langle \mathit{true}, \mathcal{A}(\square \varphi), \mathit{acc} \rangle \wedge \mathcal{A}(\varphi) \\
\mathcal{A}(\diamond \varphi) &= \langle \mathit{true}, \mathcal{A}(\diamond \varphi), \mathit{rej} \rangle \vee \mathcal{A}(\varphi) \\
\mathcal{A}(\varphi \,\mathcal{U}\, \psi) &= \mathcal{A}(\psi) \vee (\langle \mathit{true}, \mathcal{A}(\varphi \,\mathcal{U}\, \psi), \mathit{rej} \rangle \wedge \mathcal{A}(\varphi)) \\
\mathcal{A}(\varphi \mathcal{W} \psi) &= \mathcal{A}(\psi) \vee (\langle \mathit{true}, \mathcal{A}(\varphi \mathcal{W} \psi), \mathit{acc} \rangle \wedge \mathcal{A}(\varphi))
\end{aligned}
$$

The constructions are illustrated in Figure 1.

3 Checking Traces

We present three algorithms to check whether a trace satisfies a temporal formula. All are based on alternating automata, but make different optimizations and are favored in different situations.

The first algorithm attempts to match the trace with the automaton by recursively traversing the automaton in a depth-first manner. The second algorithm attempts to match the trace with the automaton by traversing the automaton in a breadth-first manner: for each element in the trace it creates all possible successor states. The third algorithm traverses the trace and automaton backwards, in which case no search has to be performed. This last approach is essentially the same as reported by Rosu and Havelund in [RH01].

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

To check whether the trace satisfies a temporal formula, we first generate the alternating automaton for the formula and then check the trace against the automaton as follows:

$$\begin{aligned}
\textsc{ct}(\mathcal{A}_1 \wedge \mathcal{A}_2, \mathit{trace}, n) &= \textsc{ct}(\mathcal{A}_1, \mathit{trace}, n) \wedge \textsc{ct}(\mathcal{A}_2, \mathit{trace}, n) \\
\textsc{ct}(\mathcal{A}_1 \vee \mathcal{A}_2, \mathit{trace}, n) &= \textsc{ct}(\mathcal{A}_1, \mathit{trace}, n) \vee \textsc{ct}(\mathcal{A}_2, \mathit{trace}, n) \\
\textsc{ct}(\langle \nu, \delta, f \rangle, \mathit{trace}, n) &= \mathit{trace}[n] \vDash \nu \ \wedge \textsc{ct}(\delta, \mathit{trace}, n+1)
\end{aligned}$$

For finite traces with length ℓ, we make the following modification to the above algorithm:

$$\begin{aligned}
&\textsc{ct}(\langle \nu, \delta, f \rangle, \mathit{trace}, n) = \mathit{trace}[n] \vDash \nu \ \wedge \textsc{ct}(\delta, \mathit{trace}, n+1) \quad n < \ell \\
&\textsc{ct}(\langle \nu, \delta, \mathit{fin} \rangle, \mathit{trace}, \ell) = \mathit{trace}[n] \vDash \nu \\
&\textsc{ct}(\langle \nu, \delta, \mathit{acc} \rangle, \mathit{trace}, \ell) = \mathit{true} \\
&\textsc{ct}(\langle \nu, \delta, \mathit{rej} \rangle, \mathit{trace}, \ell) = \mathit{false}
\end{aligned}$$

Thus a trace is accepted only if all its eventualities have been fulfilled. It is somewhat harsh to reject a trace because its last state is not accepting. Indeed the trace may have been cut off just before the eventuality would have been fulfilled. In Section 4 we present an algorithm that keeps statistics about the trace and returns a more informative answer than just acceptance or rejection.

The depth-first traversal algorithm is the easiest to implement as it follows directly the structure of the alternating automaton as generated. Its main disadvantage is that parts of the trace may be traversed multiple times (up to ℓ times where ℓ is the length of the trace). For example this occurs in checking a formula of the form

$$\Box \Diamond \varphi$$

on a trace where φ is satisfied only at the last element of the trace. At each state the algorithm will traverse the remainder of the trace to look for φ. Therefore this algorithm becomes prohibitively slow for long traces.

3.2 Breadth-first traversal

An alternative approach is to search for a run dag in a breadth-first manner, which avoids multiple traversals. A run dag $\langle V_0, V, E, \mu \rangle$ can be represented as a sequence of slices $S_0, S_1, \ldots$ where $S_i \subseteq V$, and S_i contains a dag node n

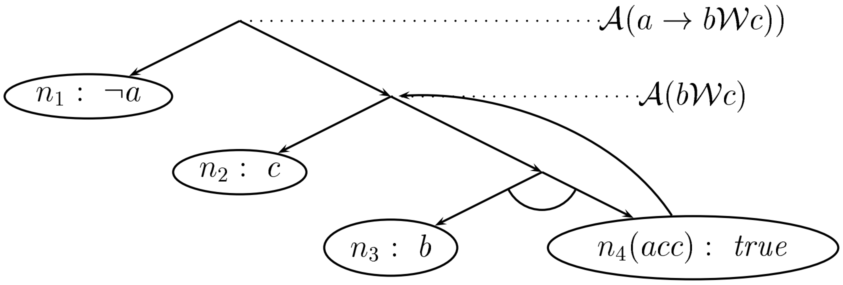

Fig. 2. Alternating automaton for $a \to b\mathcal{W}c$

iff there is a path of length i from some root node to n. This representation is particularly useful for memoryless runs.

Definition 5 (Memoryless run) A run dag is called a *memoryless run* if no automaton node occurs on two different dag nodes of the same slice.

Obviously not all runs are memoryless; however, given an arbitrary accepting run it is always possible to construct an accepting memoryless run on the same sequence of states. For trace checking it is therefore sufficient to compute memoryless runs. In the following we will call the set of automaton nodes that label the elements of a slice S_i the *configuration* C_i. The existence of a memoryless run for a trace can be checked by generating the set of possible configurations for each position of the trace. For the initial position this set is given as follows:

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

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

The set of possible configurations for the initial position of the automaton for $a \to b\mathcal{W}c$, shown in Figure 2 is given by

$$\theta(\mathcal{A}_{a \to b\mathcal{W}c}) = \{\{n_1\}, \{n_2\}, \{n_3, n_4\}\}$$

□

To check whether a trace satisfies an LTL formula φ, the algorithm generates the alternating automaton $\mathcal{A}_\varphi$ for φ and initializes the configuration set $S = \theta(\mathcal{A}_\varphi)$. Then for each element in the trace it executes the following steps:

`for` $i := 1$ `to` $|trace| - 1$

```
        S' := ∅
        for each configuration C ∈ S:
              if state-satisfied(C, trace[i]):
                    add successors(C) to S'
S := S'
```

where *state-satisfied*$(C, trace[i])$ is true if for all nodes $n:\ \langle \nu, \epsilon_{\mathcal{A}}, f\rangle$ in C

$$trace[i]\ \models\ \nu$$

is valid. *successors*(C) returns the crossproduct of all successor sets of nodes in C, that is

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

The trace is accepted if at the last position some configuration is state-satisfied and contains only accepting and final nodes.

This algorithm clearly traverses the trace only once. However it may have to 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 5.

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 Rosu and Havelund [RH01] by traversing the trace backwards. In this case traversal becomes deterministic and only one set of nodes has to be maintained at each level.

The algorithm is very similar in structure to that of depth-first traversal except that in this case we refer to previously computed values rather than making a recursive call to the checking function:

$$\mathrm{V}(\mathcal{A}_1 \wedge \mathcal{A}_2, trace, n) = \mathrm{V}(\mathcal{A}_1, trace, n) \wedge \mathrm{V}(\mathcal{A}_2, trace, n)$$

$$\mathrm{V}(\mathcal{A}_1 \vee \mathcal{A}_2, trace, n) = \mathrm{V}(\mathcal{A}_1, trace, n) \vee \mathrm{V}(\mathcal{A}_2, trace, n)$$

$$\mathrm{V}(\langle \nu, \delta, f\rangle, trace, n) = trace[n] \models \nu\ \wedge \mathrm{V}(\delta, trace, n+1)$$

We initialize with the values for ℓ, where ℓ is the length of the trace:

$$\mathrm{V}(\mathcal{A}_1 \wedge \mathcal{A}_2, trace, \ell) = \mathrm{V}(\mathcal{A}_1, trace, \ell) \wedge \mathrm{V}(\mathcal{A}_2, trace, \ell)$$

$$\mathrm{V}(\mathcal{A}_1 \vee \mathcal{A}_2, trace, \ell) = \mathrm{V}(\mathcal{A}_1, trace, \ell) \vee \mathrm{V}(\mathcal{A}_2, trace, \ell)$$

$$\mathrm{V}(\langle \nu, \delta, fin\rangle, trace, \ell) = trace[\ell] \models \nu$$

$$\mathrm{V}(\langle \nu, \delta, acc\rangle, trace, \ell) = true$$

$$\mathrm{V}(\langle \nu, \delta, rej\rangle, trace, \ell) = false$$

4 Collecting Statistics over Traces

Trace checking computes a Boolean value: *true* if the trace has an accepting run, *false* otherwise. We now generalize the analysis to return a value from some arbitrary lattice that expresses statistical information about the runs of the trace. We assume that statistics are stored in a data type $\mathcal{S}$ with a meet operation $\underline{\wedge}$, a join operation $\underline{\vee}$, a bottom element $\perp$, an operation *initial* that returns for an automaton node n the initial statistic for a traversal starting in n, and an operation *update* that returns, for a statistic and an automaton node n the new statistic after the node n is traversed. Trace checking is a special case, with $\underline{\wedge} = \wedge, \underline{\vee} = \vee$, $\perp = \mathit{false}$, $\mathit{initial}(n) = \mathit{true}$ and $\mathit{update}(S, n) = S$.

4.1 Depth-first and reverse traversal

In a depth-first or reverse traversal statistics can be collected using the following definitions:

$$\begin{aligned}
&\textsc{stat}(\mathcal{A}_1 \wedge \mathcal{A}_2, \mathit{trace}, n) = \textsc{stat}(\mathcal{A}_1, \mathit{trace}, n) \;\; \underline{\wedge} \;\; \textsc{stat}(\mathcal{A}_2, \mathit{trace}, n)\\
&\textsc{stat}(\mathcal{A}_1 \vee \mathcal{A}_2, \mathit{trace}, n) = \textsc{stat}(\mathcal{A}_1, \mathit{trace}, n) \;\; \underline{\vee} \;\; \textsc{stat}(\mathcal{A}_2, \mathit{trace}, n)\\
&\textsc{stat}(\langle \nu, \delta, f \rangle, \mathit{trace}, n) = \text{if } \mathit{trace}[n] \vDash \nu\\
&\qquad\qquad \text{then } \mathit{update}(\textsc{stat}(\delta, \mathit{trace}, n+1), \langle \nu, \delta, f \rangle) \text{ else } \perp \qquad n < \ell\\
&\textsc{stat}(\langle \nu, \delta, \mathit{fin} \rangle, \mathit{trace}, \ell) = \text{if } \mathit{trace}[n] \vDash \nu \text{ then } \mathit{initial}(\langle \nu, \delta, \mathit{fin} \rangle) \text{ else } \perp\\
&\textsc{stat}(\langle \nu, \delta, \mathit{acc} \rangle, \mathit{trace}, \ell) = \mathit{initial}(\langle \nu, \delta, \mathit{acc} \rangle)\\
&\textsc{stat}(\langle \nu, \delta, \mathit{rej} \rangle, \mathit{trace}, \ell) = \perp
\end{aligned}$$

It may be of interest to collect statistics over all runs, including those that end in rejecting nodes. In this case the last definition is replaced by the following:

$$\textsc{stat}(\langle \nu, \delta, \mathit{rej} \rangle, \mathit{trace}, \ell) = \mathit{initial}(\langle \nu, \delta, \mathit{rej} \rangle)$$

A simple example application is to determine which subsets of the automaton nodes are visited by runs of the trace. This information can be collected with the following data type:

$$\begin{aligned}
\mathcal{S} &= 2^{2^{\mathcal{N}(\mathcal{A})}}\\
\perp &= \emptyset\\
S_1 \;\underline{\vee}\; S_2 &= S_1 \cup S_2\\
S_1 \;\underline{\wedge}\; S_2 &= \{s_1 \cup s_2 \mid s_1 \in S_1, s_2 \in S_2\}\\
\mathit{initial}(n) &= \{\{n\}\}\\
\mathit{update}(S, n) &= \{s \cup \{n\} \mid s \in S\}
\end{aligned}$$

4.2 Breadth-first traversal

The gathering of statistical information in a depth-first or reverse traversal can be seen as a labeling of each node in a run with a statistical summary of the subgraph starting in that node. When statistics are collected in a breadth-first traversal, the statistical information is associated with configurations, summarizing the run up to the current slice. The initial set of annotated configurations is defined as follows.

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

$\otimes$ denotes the crossproduct:

$$\begin{aligned}
\{\langle C_1, S_1\rangle, \ldots, \langle C_n, S_n\rangle\} \otimes \{\langle C'_1, S'_1\rangle, \ldots, \langle C'_m, S'_m\rangle\} \ = \\
\left\{\langle C_i \cup C'_j, S_i \mathbin{\underline{\triangle}} S'_j\rangle \mid i = 1 \ldots n, j = 1 \ldots m\right\}
\end{aligned}$$

The algorithm generates the alternating automaton $\mathcal{A}_\varphi$ for φ and initializes the configuration set as $\theta(\mathcal{A}_\varphi)$. Then, for each element in the trace, it computes the successors of the state-satisfied configurations. The successors of a configuration are given as the following crossproduct:

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

where

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

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.

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 past (indicated by "$\leftarrow$") or future (indicated by "$\rightarrow$"). The definition of a run of such an alternating automaton reflects the presence of past nodes as follows:

Definition 6 (Run) Given an infinite sequence of states $\sigma : s_0, s_1, \ldots$, and a position $j \geq 0$, a labeled dag $\langle V_0, V, E, \mu \rangle$ is called a run of σ at position j if one of the following holds:

$$
\begin{array}{lll}
\mathcal{A} = \epsilon_{\mathcal{A}} & \text{and} & V_0 = \emptyset \\
\mathcal{A} = n & \text{and} & s_0 \vDash \nu(n) \text{ and} \\
 & & \text{there is a node } m \in V_0 \text{ s.t. } \mu(m) = n \text{ and} \\
 & & \left\{\begin{array}{ll}
 \text{(a)} & \langle E(m), V, E, \mu \rangle \text{ is a run of } \sigma \text{ in } \delta(n) \\
 & \quad \text{at } j+1, \text{if } g(n) = \rightarrow, \text{ or} \\
 \text{(b)} & \langle E(m), V, E, \mu \rangle \text{ is a run of } \sigma \text{ in } \delta(n) \\
 & \quad \text{at } j-1, \text{if } g(n) = \leftarrow \text{ and } j > 0, \text{ or} \\
 \text{(c)} & E(m) = \emptyset \text{ if } g(n) = \leftarrow, f(n) = acc, \\
 & \quad \text{and } j = 0.
 \end{array}\right. \\
\mathcal{A} = \mathcal{A}_1 \wedge \mathcal{A}_2 & \text{and} & \langle V_0, V, E \rangle \text{ is a run in } \mathcal{A}_1 \text{ and} \\
 & & \langle V_0, V, E \rangle \text{ is a run in } \mathcal{A}_2 \\
\mathcal{A} = \mathcal{A}_1 \vee \mathcal{A}_2 & \text{and} & \langle V_0, V, E \rangle \text{ is a run in } \mathcal{A}_1 \text{ or} \\
 & & \langle V_0, V, E \rangle \text{ is a run in } \mathcal{A}_2
\end{array}
$$

Given an LTL formula φ, an alternating automaton $\mathcal{A}(\varphi)$ is constructed as before, where all nodes constructed before are future nodes, and with the following additions for the past operators:

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

Example
For a causality formula $\varphi : \square(p \rightarrow \diamondminus r)$ with p, q, and r state formulas, the automaton is shown in Figure 3. $\square$

The depth-first algorithm can be extended to include past operators by

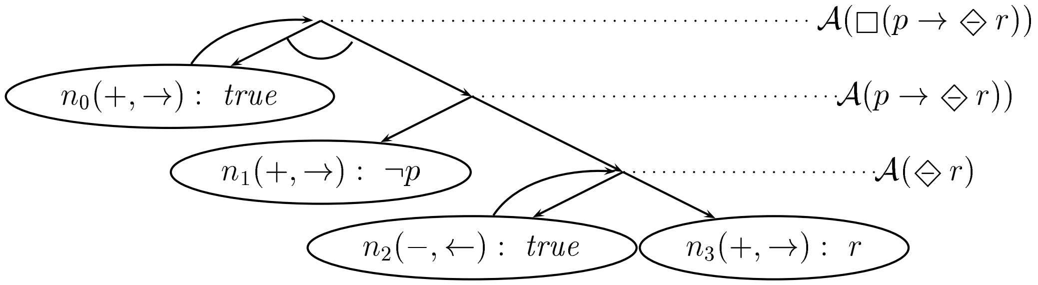

Fig. 3. Alternating automaton for $\Box(p \to \diamondminus r)$

simply adding the following calls:

$$\textsc{ct}(\langle \nu, \delta, f, \leftarrow \rangle, \mathit{trace}, n) = \mathit{trace}[n] \models \nu \ \wedge \textsc{ct}(\delta, \mathit{trace}, n-1) \quad n > 0$$
$$\textsc{ct}(\langle \nu, \delta, \mathit{acc}, \leftarrow \rangle, \mathit{trace}, 0) = \mathit{trace}[n] \models \nu$$
$$\textsc{ct}(\langle \nu, \delta, \mathit{rej}, \leftarrow \rangle, \mathit{trace}, 0) = \mathit{false}$$

For the breadth-first algorithm the situation is more complicated. Slice S_i now contains all nodes that are reached from some root node on a path with m future nodes and n past nodes such that $m - n = i$. Two modifications are necessary in the computation of successors of a configuration C:

(i) A successor configuration C' may contain nodes that are reached through a past node in the successor configuration of C'. Such nodes must thus be children of past nodes: *past-children*$(\mathcal{A}) = \bigcup_{n \in \mathcal{PN}(\mathcal{A})} \rho(n)$ where

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

(ii) For all past nodes $\langle \nu, \delta, f, \leftarrow \rangle$ in a successor configuration C', the automaton δ must be satisfied in C. Let $C \models \mathcal{A}$ denote that one of the following holds:

$$\begin{array}{lll}
\mathcal{A} = \epsilon_{\mathcal{A}} & & \\
\mathcal{A} = n & \text{and} & n \in C \\
\mathcal{A} = \mathcal{A}_1 \wedge \mathcal{A}_2 & \text{and} & C \models \mathcal{A}_1 \text{ and } C \models \mathcal{A}_2 \\
\mathcal{A} = \mathcal{A}_1 \vee \mathcal{A}_2 & \text{and} & C \models \mathcal{A}_1 \text{ or } C \models \mathcal{A}_2
\end{array}$$

Hence, the successor configurations are now computed as follows:

$$\begin{aligned} \mathit{successors}\ (C) = \{\ C' \in (\textstyle\bigotimes_{n\in C\cap\mathcal{FN}(\mathcal{A})}\theta(\delta(n)) \otimes \mathcal{P}(\mathit{past\text{-}children}(\mathcal{A}))) \\ \mid\ \text{for all } n' \in C' \cap \mathcal{PN}(\mathcal{A})\ .\ C \vDash \delta(n')\ \} \end{aligned}$$

where $\mathcal{P}$ denotes the power set. The configuration set is initialized as

$$\mathit{initial}(\theta(\mathcal{A}_\varphi) \otimes \mathcal{P}(\mathit{past\text{-}children}(\mathcal{A})))$$

where *initial*(S) returns exactly those configurations in S that do not contain any rejecting past nodes. The trace is accepted if at the last position some configuration is state-satisfied and does not contain any rejecting future nodes.

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 [BBC+00]. 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 consisting of 10% a's, 40% b's, 25% c's, and 25 % d's. For φ_1 a "z" was added to the end of the trace and for φ_2 and φ_3 an "a" was added to the end to ensure satisfaction for easier comparison. The running times are presented in Tables 1, 2, 3. The program was run on a Sony VAIO laptop with an 850 MHz Pentium III processor, 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 every tenth trace element is an "a"), 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

trace length	depth-first	breadth-first	reverse traversal
1000	1733	78	23
2000	8402	54	27
3000	21940	76	36
4000	44185	99	44
5000	76899	123	55

Fig. 4. Running times in milliseconds for checking $\Box \Diamond z$

trace length	depth-first	breadth-first	reverse traversal
1000	40	82	24
2000	79	52	27
3000	112	73	36
4000	222	94	45
5000	247	117	56

Fig. 5. Running times in milliseconds for checking $\Box \Diamond a$

trace length	depth-first	breadth-first	reverse traversal
1000	45	876	48
2000	87	1660	96
3000	185	2377	138
4000	217	3244	181
5000	298	4034	225

Fig. 6. Running times in milliseconds for checking $\Box(b \rightarrow \neg a \,\mathcal{U}\, (a \,\mathcal{U}\, (\neg a \,\mathcal{U}\, a)))$

the depth-first algorithm is comparable to that of the reverse traversal.

Based on these, very 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.

Acknowledgements

We thank Klaus Havelund and Grigore Rosu 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.

References

[BBC+00] N.S. Bjørner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H.B. Sipma, and T.E. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design*, 16(3):227–270, June 2000.

[Dru00] D. Drusinsky. The Temporal Rover and the ATG Rover. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification, $7^t h$ International SPIN Workshop*, vol. 1885 of *LNCS*, pages 323–330. Springer-Verlag, 2000.

[Hav00] K. Havelund. Using runtime analysis to guide model checking of java programs. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification, $7^t h$ International SPIN Workshop*, vol. 1885 of *LNCS*, pages 245–264. Springer-Verlag, 2000.

[HR01] K. Havelund and G. Rosu. Testing linear temporal logic formula on finite execution traces. 2001. Submitted for publication.

[MP87] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by ∀-automata. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, number 398 in LNCS, pages 124–164. Springer-Verlag, Berlin, 1987. Also in *Proc. 14th ACM Symp. Princ. of Prog. Lang.*, Munich, Germany, pp. 1–12, January 1987.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[MS00] Z. Manna and H.B. Sipma. Alternating the temporal picture for safety. In U. Montanari, J.D. Rolim, and E. Welzl, editors, *Proc. 27th Intl. Colloq. Aut. Lang. Prog.*, vol. 1853, pages 429–450, Geneva, Switzerland, July 2000. Springer-Verlag.

[RH01] G. Rosu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. 2001. Submitted for publication.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, pages 133–191. Elsevier Science Publishers (North-Holland), 1990.

[Var96] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency. Structure versus Automata*, vol. 1043 of *LNCS*, pages 238–266. Springer-Verlag, 1996.

[Var97] M.Y. Vardi. Alternating automata: Checking truth and validity for temporal logics. In *Proc. of the* 14^{th} *Intl. Conference on Automated Deduction*, vol. 1249 of *LNCS*. Springer-Verlag, July 1997.