# Efficient Model Checking Via Büchi Tableau Automata*

Girish S. Bhat[1], Rance Cleaveland[2], and Alex Groce[3]

[1] Cosine Communications, Inc. girish23@hotmail.com
[2] Department of Computer Science, SUNY at Stony Brook, rance@cs.sunysb.edu
[3] School of Computer Science, Carnegie-Mellon University, agroce+@cs.cmu.edu

**Abstract.** This paper describes an approach to engineering efficient model checkers that are generic with respect to the temporal logic in which system properties are given. The methodology is based on the "compilation" of temporal formulas into variants of alternating tree automata called *alternating Büchi tableau automata* (ABTAs). The paper gives an efficient on-the-fly model-checking procedure for ABTAs and illustrates how translations of temporal logics into ABTAs may be concisely specified using inference rules, which may be thus seen as high-level definitions of "model checkers" for the logic given. Heuristics for simplifying ABTAs are also given, as are experimental results in the CWB-NC verification tool suggesting that, despite the generic ABTA basis, our approach can perform better than model checkers targeted for specific logics. The ABTA-based approach we advocate simplifies the retargeting of model checkers to different logics, and it also allows the use of "compile-time" simplifications on ABTAs that improves model-checker performance.

## 1 Introduction

Temporal-logic model-checking algorithms determine whether or not a given system's behavior conforms to requirements formulated as properties in an appropriate temporal logic. Numerous algorithms for different logics and system modeling formalisms have been developed and implemented [2, 6, 8, 9, 14, 20, 24, 25, 27], and case studies have demonstrated the utility of the technology (see [10] for a survey).

Traditional model checkers work for one logic and one class of system models. For example, the algorithm in [9] checks whether systems given as Kripke structures obey properties expressed in CTL, while the automaton-based approach of [29] works on Kripke structures and properties given in linear-time temporal logic. Other algorithms have been developed in the context of labeled transition systems and the modal mu-calculus [14], Modecharts and real-time logics [32], and so on. This paradigm for model checking has yielded great research insights, but it has the disadvantage that changes to the modeling formalism (e.g. by changing the interpretation of state and transition labels) or the logic (e.g. by introducing domain-specific operators) necessitate a redesign and reimplementation of the relevant model-checking algorithm. The amount of work needed to "retarget" a model checker can be an important factor hampering the uptake of the technology.

The goal of this paper is to demonstrate the utility of an alternative view of model checking that relies on translating temporal formulas into intermediate structures, *alternating Büchi tableau automata* (ABTA) [6], that a model checker then works on. ABTAs are variants of alternating tree automata [23] that support efficient model checking while enabling various "compile-time" optimizations to be performed. They also support the abstract definition, via "proof rules," of translation procedures for different temporal logics. By factoring out the formulation of model-checking questions from the routines that answer them, our framework simplifies retargeting model checkers to different system formalisms and temporal logics.

The remainder of this paper develops as follows. The next section presents the system models considered in this paper and defines ABTAs. Section 3 then develops an efficient on-the-fly model-checking algorithm for a large class of ABTAs, and the section following describes simplifications that may be performed on ABTAs. A method for translating temporal logics into ABTAs is given via an extended example in Section 5, and the section following describes an implementation and experimental results. Section 7 discusses related work, while the final section contains our conclusions and future work. An appendix contains full pseudo-code for the model-checking algorithm.

* Research supported by US Air Force Office of Scientific Research grant F49620-95-1-0508; US Army Research Office grants P-38682-MA, DAAD190110003,and DAAD190110019; and US National Science Foundation grants CCR-9257963, CCR-9505562, CCR-9996086, and CCR-9988489.

## 2 Transition Systems and Tableau Automata

This section defines our system models and introduces alternating Büchi tableau automata. In what follows we fix disjoint sets $(p, p', p_1, \ldots \in)\mathcal{A}$ and $(\theta, \theta', \theta_1, \ldots \in)\mathcal{A}_{\text{act}}$ of atomic *state* and *action* propositions, respectively.

### 2.1 Transition Systems

*Transition systems* encode the operational behavior of systems.

**Definition 1** A transition system (TS) is a tuple $\langle S, A, \ell_S, \ell_A, \longrightarrow, s_I \rangle$ where $S$ is a set of states; $A$ is a set of actions; $\ell_S : S \longrightarrow 2^{\mathcal{A}}$ is the *state labeling function*; $\ell_A : A \longrightarrow 2^{\mathcal{A}_{\text{act}}}$ is the *action labeling function*; $\longrightarrow \subseteq S \times A \times S$ is the *transition relation*; and $s_I$ is the *start state*. ∎

Intuitively, $S$ contains the states a system may enter and $A$ the atomic actions a system may perform. The labeling functions $\ell_S$ and $\ell_A$ indicate which atomic propositions hold of a given state or action, while $\longrightarrow$ encodes the execution steps the system may engage in and $s_I$ the initial state of the system. We write $s \xrightarrow{\alpha} s'$ in lieu of $\langle s, \alpha, s' \rangle \in \longrightarrow$.

**Definition 2** Let $\mathcal{T} = \langle S, A, \ell_S, \ell_A, \longrightarrow, s_I \rangle$ be a TS.

1. A *transition sequence* from $s_0 \in S$ is a sequence $\sigma = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_k} s_k$, where $0 \leq k \leq \infty$. We define the *length* of $\sigma$, $|\sigma|$, to be $k$. If $|\sigma| = \infty$ we call $\sigma$ *infinite*; otherwise, it is *finite*.
2. An *execution* from $s_0$ is a maximal transition sequence from $s_0$, that is, a sequence $\sigma$ with the property that either $|\sigma| = \infty$, or $|\sigma| < \infty$ and $s_{|\sigma|} \not\xrightarrow{\alpha} s'$ for any $\alpha \in A$ and $s' \in S$.

If $s \in S$ then we use $\mathcal{E}_{\mathcal{T}}(s)$ to denote the set of executions in $\mathcal{T}$ from $s$. ∎

### 2.2 Alternating Büchi Tableau Automata

In this paper we use *alternating Büchi tableau automata* (ABTAs) as an intermediate representation for system properties. ABTAs are alternating tree automata, although they differ in subtle and noteworthy ways from the automata introduced in [23]; Section 7 gives details. To define ABTAs formally we first introduce the following syntactic sets. Let $\neg$ be a distinguished negation symbol; we define $\mathcal{L} = \mathcal{A} \cup \{\, \neg p \mid p \in \mathcal{A} \,\}$ to be the set of state literals and $\mathcal{L}_{\text{act}} = \mathcal{A}_{\text{act}} \cup \{\, \neg\theta \mid \theta \in \mathcal{A}_{\text{act}} \,\}$ to be the set of action literals. We also use $\Theta, \Theta', \ldots$ to range over subsets of $\mathcal{L}_{\text{act}}$. ABTAs may now be defined as follows.

**Definition 3** An *alternating Büchi tableau automaton* (ABTA) is a tuple $\langle Q, \ell, \longrightarrow, q_I, \mathcal{F} \rangle$, where $Q$ is a finite set of *states*; $\ell : Q \to \mathcal{L} \cup \{\neg, \wedge, \vee, [\Theta], \langle\Theta\rangle\}$ is the *state labeling*; $\longrightarrow \subseteq (Q \times Q)$, the *transition relation*, satisfies the condition below for all $q \in Q$; $q_I \in Q$ is the start state; and $\mathcal{F} \subseteq 2^S$ is the acceptance condition. The additional condition $\to$ must satisfy is:

$$|\{\, q' \mid q \to q' \,\}| \begin{cases} = 0 \text{ if } \ell(q) \in \mathcal{L} \\ \geq 1 \text{ if } \ell(q) \in \{\wedge, \vee\} \\ = 1 \text{ if } \ell(q) \in \{\neg, \langle\Theta\rangle, [\Theta]\} \end{cases}$$

∎

As ABTAs are special node-labeled graphs we use typical graph-theoretic notions, including cycle, path, strongly-connected component, etc. We also write $q \to^* q'$ if there exists a path from $q$ to $q'$ in ABTA $\mathcal{B}$. We say that an ABTA is *well-formed* if, whenever $\ell(q) = \neg$, then $q$ does not appear on a cycle of $\to$ edges. We only consider well-formed ABTAs in what follows.

Besides alternating tree automata, ABTAs may be viewed as abstract syntax for a fragment of the mu-calculus [6]. They may also be seen as defining system properties in terms of how the property in question is to be "proved", and we develop this intuition in presenting their semantics. More specifically, an ABTA defines a property of transition systems by encoding a "proof schema" for establishing that the property holds for a transition system. The states in the ABTA can be seen as goals, with the labels in the states defining the relationship that must hold between a state and its "subgoals". So if one wishes to show that a transition-system state $s$ "satisfies" a state $q$ in an ABTA, and the label of $q$ is $\wedge$, then one must show that $s$ satisfies each of $q$'s children. The $[\Theta]$ and $\langle\Theta\rangle$ labels correspond to single-step modalities; for a transition-system state $s$ to satisfy an ABTA state $q$ whose label is $[\Theta]$, one must show that for each $s'$ such that $s \xrightarrow{\alpha} s'$ for some $\alpha$ "satisfying" $\Theta$, $s'$ must satisfy the (unique) successor of $q$. Finally, the acceptance sets enable "proofs" to be infinite: an "infinite positive" proof is deemed valid if every "path" in the proof "touches" each set in $F$ infinitely often, while an infinite "negative proof" is valid if it fails to "touch" at least one set in $F$ infinitely often. (The first clause is the same as the generalized Büchi acceptance condition defined in [15]. It should also be noted that the second clause indicates that ABTAs have a "co-Büchi" component to their acceptance condition.) These intuitions may be formalized in terms of "runs" of an ABTA. To define these we first introduce the following terminology.

**Definition 4** Let $\mathcal{T} = \langle S, A, \ell_S, \ell_A, \longrightarrow, s_I\rangle$ be a TS with $s \in S$.

1. Let $p \in \mathcal{A}$. Then $s \models_{\mathcal{T}} p$ if and only if $p \in \ell_S(s)$, and $s \models_{\mathcal{T}} \neg p$ if and only if $p \notin \ell_S(s)$.
2. Let $\theta \in \mathcal{A}_{\text{act}}$. Then $\alpha \models_{\mathcal{T}} \theta$ if and only if $\theta \in \ell_A(\alpha)$, and $\alpha \models_{\mathcal{T}} \neg\theta$ if and only if $\theta \notin \ell_A(\alpha)$.
3. Let $\Theta \subseteq \mathcal{L}_{\text{act}}$. Then $\alpha \models_{\mathcal{T}} \Theta$ if and only if $\alpha \models \theta$ for every $\theta \in \Theta$. We write $s \xrightarrow{\Theta} s'$ if and only if $s \xrightarrow{\alpha} s'$ for some $\alpha \in A$ such that $\alpha \models_{\mathcal{T}} \Theta$ and $s \overset{\Theta}{\not\longrightarrow}$ if there is no $s'$ such that $s \xrightarrow{\Theta} s'$.

∎

**Definition 5** A *run* of an ABTA $\mathcal{B} = \langle Q, \ell, \rightarrow_{\mathcal{B}}, q_I, \mathcal{F}\rangle$ on a TS $\mathcal{T} = \langle S, A, \ell_S, \ell_A, \rightarrow_{\mathcal{T}}, s_I\rangle$ is a maximal tree in which the nodes are classified as *positive* or *negative* and are labeled by elements of $Q \times S$ as follows.

- The root of the tree is a positive node and is labeled with $\langle q_I, s_I\rangle$.
- If $\sigma$ is a positive (negative) node with label $\langle q, s\rangle$ such that $\ell(q) = \neg$ and $q \rightarrow_{\mathcal{B}} q'$, then $\sigma$ has one negative (positive) child labeled $\langle q', s\rangle$.
- Otherwise, for a positive node $\sigma$ labeled with $\langle q, s\rangle$:
  - If $\ell(q) \in \mathcal{L}$ then $\sigma$ is a leaf.
  - If $\ell(q) = \wedge$ and $\{\, q' \mid q \rightarrow_{\mathcal{B}} q' \,\} = \{q_1, ..q_m\}$, then $\sigma$ has positive children $\sigma_1, .., \sigma_m$, with $\sigma_i$ labeled by $\langle q_i, s\rangle$.
  - If $\ell(q) = \vee$ then $\sigma$ has one positive child, $\sigma'$, and $\sigma'$ is labeled with $\langle q', s\rangle$ for some $q' \in \{\, q' \mid q \rightarrow_{\mathcal{B}} q' \,\}$.
  - If $\ell(q) = [\Theta]$ , $q \rightarrow q'$, and $\{\, s' \mid s \xrightarrow{\Theta}_{\mathcal{T}} s' \,\} = \{s_1, .., s_m\}$ then $\sigma$ has positive children $\sigma_1, .., \sigma_m$, with $\sigma_i$ is labeled by $\langle q', s_i\rangle$.
  - If $\ell(q) = \langle\Theta\rangle$ and $q \rightarrow q'$ then $\sigma$ has one positive child $\sigma'$, and $\sigma'$ is labeled by $\langle q', s'\rangle$ for some $s'$ such that $s \xrightarrow{\Theta}_{\mathcal{T}} s'$.
- Otherwise, for a negative node $\sigma$ labeled with $\langle q, s\rangle$:
  - If $\ell(q) \in \mathcal{L}$ then $\sigma$ is a leaf.
  - If $\ell(q) = \wedge$ then $\sigma$ has one negative child labeled with $\langle q', s\rangle$ for some $q' \in \{\, q' \mid q \rightarrow_{\mathcal{B}} q' \,\}$.
  - If $\ell(q) = \vee$ and $\{\, q' \mid q \rightarrow_{\mathcal{B}} q' \,\} = \{q_1, ..q_m\}$, then $\sigma$ has negative children $\sigma_1, .., \sigma_m$, with $\sigma_i$ labeled by $\langle q_i, s\rangle$.
  - If $\ell(q) = [\Theta]$ and $q \rightarrow_{\mathcal{B}} q'$ then $\sigma$ has one negative child $\sigma'$ labeled by $\langle q', s'\rangle$ for some $s'$ such that $s \xrightarrow{\Theta}_{\mathcal{T}} s'$.
  - If $\ell(q) = \langle\Theta\rangle$ , $q \rightarrow_{\mathcal{B}} q'$, and $\{\, s' \mid s \xrightarrow{\Theta}_{\mathcal{T}} s' \,\} = \{s_1, .., s_m\}$ then $\sigma$ has negative children $\sigma_1, .., \sigma_m$, with $\sigma_i$ is labeled by $\langle q', s_i\rangle$.

∎

In a well-formed ABTA, every infinite path has a suffix that contains either positive or negative nodes, but not both. Such a path is referred to as *positive* in the former case and *negative* in the latter. We now define the notion of *success* of a run.

**Definition 6** Let $R$ be a run of ABTA $\mathcal{B} = \langle Q, \ell, \rightarrow_{\mathcal{B}}, q_I, \mathcal{F}\rangle$ on a TS $\mathcal{T} = \langle S, A, \ell_S, \ell_A, \rightarrow_{\mathcal{T}}, s_I\rangle$.

1. A *positive leaf* labeled $\langle q, s\rangle$ is *successful* if and only if $s \models_{\mathcal{T}} \ell(q)$ or $\ell(q) = [\Theta]$ and $s \overset{\Theta}{\not\longrightarrow}_{\mathcal{T}}$.

2. A *negative leaf* is *successful* if and only if $s \not\models_{\mathcal{T}} \ell(q)$ or $\ell(q) = \langle \Theta \rangle$ and $s \not\xrightarrow{\Theta}_{\mathcal{T}}$.
3. A *positive path* is *successful* if and only if for each $F \in \mathcal{F}$ some $q \in F$ occurs infinitely often.
4. A *negative path* is *successful* if and only if for some $F \in \mathcal{F}$ there is no $q \in F$ that occurs infinitely often.

Run $R$ is *successful* if and only if every leaf and every infinite path in $R$ is successful. TS $\mathcal{T}$ *satisfies* $\mathcal{B}$ ($\mathcal{T} \models \mathcal{B}$) if and only if there exists a successful run of $\mathcal{B}$ on $\mathcal{T}$. ∎

It is straightforward to establish the following, where if $\mathcal{B}$ is an ABTA with state $q$ then $\mathcal{B}[q]$ is the ABTA $\mathcal{B}$ with the start state changed to $q$.

**Lemma 1.** *Let $\mathcal{T}$ be a TS, let $\mathcal{B} = \langle Q, \ell, \longrightarrow_{\mathcal{B}}, q_I, \mathcal{F} \rangle$ be an ABTA, and let $q, q' \in Q$ be such that $q \longrightarrow_{\mathcal{B}} q'$ and $\ell(q) = \neg$. Then $\mathcal{T} \models \mathcal{B}[q]$ if and only if $\mathcal{T} \not\models \mathcal{B}[q']$.*

Next we define the subset of *and-restricted* ABTAs.

**Definition 7** ABTA $\langle Q, \ell, \rightarrow, q_I, \mathcal{F} \rangle$ is *and-restricted* if and only if every $q \in Q$ satisfies:

1. if $\ell(q) = \wedge$ then there is at most one $q'$ such that $q \rightarrow q'$ and $q' \rightarrow^* q$; and
2. if $\ell(q) = [\Theta]$ and $q \rightarrow q'$ then $q' \not\rightarrow^* q$. ∎

And-restriction plays an important role in our model-checking procedure, and we comment more on it here. In an and-restricted ABTA the strongly-connected component of a state labeled by $\wedge$ can contain at most one of the state's children; a state labeled by $[\Theta]$ on the other hand is guaranteed to belong to a different strongly-connected component that its child. And-restrictedness differs from the notion of *hesitation* introduced in [23]; an ABTA would be hesitant if, roughly speaking, every strongly-connected component of a node labeled by $\wedge$ or $[\Theta]$ would contain only nodes labeled by $\wedge$ or $\Theta$. Nevertheless, and-restrictedness plays the same role in our theory that hesitation does in [23]: automata obeying these conditions give rise to more efficient model-checking routines while still providing sufficient expressiveness to encode logics such as CTL*.

## 3 ABTAs and Model Checking

Checking whether or not $\mathcal{T} \models \mathcal{B}$ for TS $\mathcal{T}$ and ABTA $\mathcal{B}$ reduces to searching for the existence of a successful run of $\mathcal{B}$ on $\mathcal{T}$. This section presents an efficient on-the-fly algorithm for this check in the setting of and-restricted ABTAs.

### 3.1 TSs, ABTAs and Product Graphs

Our ABTA model-checking algorithm works by exploring the "product graph" of an ABTA and a TS. In what follows, fix ABTA $\mathcal{B} = \langle Q, \ell, \longrightarrow_{\mathcal{B}}, q_I, \mathcal{F} \rangle$ and TS $\mathcal{T} = \langle S, A, \ell_S, \ell_A, \longrightarrow_{\mathcal{T}}, s_I \rangle$, and assume that $\mathcal{F} = \{F_0, \ldots F_{n-1}\}$. The *product graph* of $\mathcal{B}$ and $\mathcal{T}$ has vertex set $V = Q \times S \times \{0, \ldots, n-1\}$ and edges $E \subseteq V \times V$ defined by $(\langle q, s, i \rangle, \langle q', s', i' \rangle) \in E$ if and only if:

- there exist nodes $\sigma$ and $\sigma'$ in some run of $\mathcal{B}$ on $\mathcal{T}$ labeled $\langle q, s \rangle$ and $\langle q', s' \rangle$ respectively and such that $\sigma \rightarrow \sigma'$; and
- either $q \notin F_i$ and $i' = i$, or $q \in F_i$ and $i' = (i+1) \bmod n$.

$E_N \subseteq E$ consists of those edges $(\langle q, s, i \rangle, \langle q', s', i' \rangle)$ such that $q$ and $q'$ are in different strongly-connected components in $\mathcal{B}$, while $E_R = E - E_N$. We sometimes refer to $E_N$ as the *nonrecursive* relation and to $E_R$ as the *recursive* relation. A vertex $\langle q, s, i \rangle$ in the product graph is said to be *accepting* if and only if $q \in \mathcal{F}_0$ and $i = 0$.

### 3.2 Searching the Product Graph

We now present an algorithm for determining if the product graph mentioned above contains a successful run in the case that the ABTA $\mathcal{B}$ is and-restricted. The routine is based on the memory-efficient on-the-fly algorithm for emptiness-checking of Büchi word automata in [15]; as is the case in that algorithm our goal is to eliminate the storage penalty associated with the "strongly-connected component" algorithms [23]. The alterations are necessitated by the fact that ABTAs contain conjunctive as well as disjunctive states and are intended to accept TSs (i.e. trees) rather than words.

Like the algorithm in [15] ours employs two depth-first searches, `DFS1` and `DFS2`, that attempt to mark nodes as either true or false. The purpose of the former is to search for true and false leaves in the product graph, and to "restart" the latter whenever an accepting node is found. The latter determines whether or not the node given to it is reachable from itself via nodes not previously traversed by `DFS2`. The success of `DFS2` has implications for the existence of runs with successful paths. Pseudo-code for the these procedures, and for the auxiliary routine `markAndPropagate`, may be found in the appendix.

When exploring $v = \langle q, s, i \rangle$, `DFS1` uses the label of $q$ in $\mathcal{B}$ and the transitions from $s$ in $\mathcal{T}$ to guide its search. The non-recursive successors of $v$ are processed first via recursive calls to `DFS1`; if the results do not immediately imply the truth or falsity of $v$, then `DFS1` is called recursively on $v$'s recursive children. (Note that this simplifies the treatment of negation: no explicit treatment of "infinite negative paths" is necessary in the algorithm. Also note that since ABTAs are and-restricted, all but one of the children of a node labeled by $\wedge$ can have their truth values determined by recursive calls to DFS1. This latter fact is crucial to the correctness of our algorithm.) If these results are inconclusive, and $v$ is accepting, then `DFS2` is called to determine if $v$ is reachable from itself. If this is the case, then $v$ is labeled as true. (DFS2 cycles involving FALSE states are, of course, not allowed).

A subtlety arises in our setting when a recursive child $v'$ of $v$ has been visited previously by `DFS1` and $v'$ has not been marked true or false. The node $v'$ cannot necessarily be assumed to be false, as is implicitly done in [15], because there may be a successful cycle in the same strongly-connected component as it that was not detected until after `DFS1(`$v'$`)` terminated. To avoid needless recomputation in this case, we maintain a *dependency* set for each node; these sets contain nodes that should become true if the indicated node is found to be true. In the example above we would insert $v$ into the dependency set of $v'$; if $v'$ is subsequently marked as true, then $v$ would be as well.

We now state the following correctness theorem.

**Theorem 8.** *DFS1($\langle q_I, s_I, 0 \rangle$) returns "true" if and only if $\mathcal{T} \models \mathcal{B}$.*

We may also characterize the time- and space-complexity as follows.

**Theorem 9.** *Let $\mathcal{B} = \langle Q, \ell, \longrightarrow_{\mathcal{B}}, q_I, \mathcal{F} \rangle$ be an ABTA and $\mathcal{T} = \langle S, A, \ell_S, \ell_A, \longrightarrow_{\mathcal{T}}, s_I \rangle$ be a TS. Then DFS1($q_I, s_I, 0$) runs in time linear in the size of the product graph of $\mathcal{B}$ and $\mathcal{T}$, whose vertex set is bounded in size by $|Q| \cdot |S| \cdot |\mathcal{F}|$, where $|\mathcal{F}|$ is the number of component sets in $\mathcal{F}$.*

## 4 Reducing ABTAs

The previous theorem indicates that the time needed to check whether or not $\mathcal{T} \models \mathcal{B}$ depends intimately on the number of states in $\mathcal{B}$. Consequently, any preprocessing that reduces the number of states in $\mathcal{B}$ can have a significant impact on model-checker performance. In this section we present several heuristics that may be used to eliminate states in ABTAs.

*Büchi State Set Minimalization* The ABTA acceptance condition specifies that an infinite (positive) path in a run is successful if and only if that path contains an infinite number of states from each of the sets of accepting states. This can only occur when a cycle in the ABTA contains at least one state from each set of accepting states. Moreover, a state not part of any such cycle can safely be removed from all member sets in $\mathcal{F}$, since no infinite path going through that state can satisfy the Büchi condition.

To check for such states we perform a depth-first search for cycles that contain at least one member of each set of accepting states. If for a particular state such a cycle does not exist, that state is removed from all accepting sets that contain it. While not reducing the size of the ABTA directly, this transformation is important for two reasons.

1. *It improves the performance of other reductions.* Some of the other reductions may only be applied to states that are members of the same accepting sets. Eliminating states from accepting sets improves their performance.
2. *The size of the product automaton is reduced.* Each state in the product graph contains an index reflecting the member set of $\mathcal{F}$ "currently" being searched for. This search procedure is unnecessary for states not having the kind of cycle just described; by removing these states from acceptance sets, unnecessary vertices associated with this search can be avoided.

*Constant Propagation* Some atomic state propositions are uniformly true or false of all TS states, and these values can be propagated upwards until no further reduction is possible.

*Associative Joining* Because $\vee$ and $\wedge$ are associative we can also apply another reduction: for any $\wedge(\vee)$-labeled state $q$ with a transition to another $\wedge(\vee)$-labeled state $q'$, where $q$ and $q'$ are in the same sets of accepting states, remove the transition from $q$ to $q'$ and add outgoing transitions from $q$ to every state to which $q'$ had a transition. This is applied recursively (if $q'$ has a transition to another $\wedge(\vee)$-labeled state $q''$ we also add its outgoing transitions to $q$, and so forth). This has two benefits: (1) the state $q'$ may become unreachable and hence removable, thereby reducing ABTA size and (2) model checking avoids passing through $q'$ (and $q''$, etc.) in the depth-first searches starting from $q$. Because $q$ and $q'$ must be in the same sets of accepting states, this simplification is much more effective when accepting-state set minimalization is first carried out.

*Quotienting via Bisimulation* The final simplification involves merging states with the same "structure." We do this using bisimulation [26]. Specifically, we alter the traditional definition of bisimulation to take account of state labels and acceptance set information, and we then quotient $\mathcal{B}$ by this equivalence. To ensure maximum reduction this should always be the last simplification applied.

## 5 Translating Temporal Formulas into ABTAs

A virtue of ABTA-based model checking is that translation procedures for temporal logics into ABTAs may be defined abstractly via "proof rules." This section illustrates this idea via an example, by giving the rules needed to translate a variant of CTL* into ABTAs. The logic, which we call *Generalized CTL** (GCTL*), extends CTL* by allowing formulas to constrain actions as well as states. While the logic itself is not very novel, it does contain "deviations" from CTL* that typically require alterations to a CTL* model checker. Our intention is to show that proof-rule-based translations, coupled with generic ABTA technology, can make it easier to define such "alterations".

The syntax for our logic is given below, where $p \in \mathcal{A}$ and $\theta \in \mathcal{A}_{\mathrm{act}}$.

$$\begin{aligned} \mathcal{S} &::= p \mid \neg p \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid \mathsf{A}\mathcal{P} \mid \mathsf{E}\mathcal{P} \\ \mathcal{P} &::= \theta \mid \neg\theta \mid \mathcal{S} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \vee \mathcal{P} \mid \mathsf{X}\mathcal{P} \mid \mathcal{P}\mathsf{U}\mathcal{P} \mid \mathcal{P}\mathsf{V}\mathcal{P} \end{aligned}$$

The formulas generated by $\mathcal{S}$ are called state formulas, while those generated by $\mathcal{P}$ are called path formulas. The state formulas constitute the formulas of GCTL*. In what follows we use $\psi, \psi', \psi_1, \ldots$ to range over state formulas and $\phi, \phi', \phi_1, \ldots$ to range over path formulas.

Semantically, the logic departs from traditional CTL* in two respects. Firstly, the paths that path formulas are interpreted over have the form $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots$ and thus contain actions as well as states. Secondly, as TSs may contain deadlocked states some provision must be made for finite maximal paths as models. The GCTL* semantics follows a standard convention in temporal logic by allowing the last state in a finite maximal path to "loop" to itself; the action component of these implicit transitions is assumed to violate all atomic action propositions $\theta \in \mathcal{A}_{\mathrm{act}}$.

Mathematically, a state satisfies $\mathsf{A}\phi$ ($\mathsf{E}\phi$) if every execution (some execution) emanating from the state satisfies $\phi$. An execution satisfies a state formula if the initial state in the execution does, and it satisfies $\theta$ if the execution contains at least one transition and the label of the first transition on the path satisfies $\theta$. A path satisfies $\neg\theta$ if either the first transition on the path is labeled by an action not satisfying $\theta$ or the path

$$R1\ \wedge : \frac{\psi_1 \wedge \psi_2}{\psi_1 \quad \psi_2} \qquad R2\ \vee : \frac{\psi_1 \vee \psi_2}{\psi_1 \quad \psi_2} \qquad R3\ \vee : \frac{\mathsf{E}(\psi)}{\psi} \qquad R4\ \neg : \frac{\neg\psi}{\psi}$$

$$R5\ \neg : \frac{\mathsf{A}(\Phi)}{\mathsf{E}(\mathsf{neg}\,\Phi)} \qquad R6\ \wedge : \frac{\mathsf{E}(\Phi,\psi)}{\mathsf{E}(\Phi) \quad \mathsf{E}(\psi)} \qquad R7\ \wedge : \frac{\mathsf{E}(\Phi,\phi_1 \wedge \phi_2)}{\mathsf{E}(\Phi,\phi_1,\phi_2)} \quad R8\ \vee : \frac{\mathsf{E}(\Phi,\phi_1 \vee \phi_2)}{\mathsf{E}(\Phi,\phi_1) \quad \mathsf{E}(\Phi,\phi_2)}$$

$$R9\ \vee : \frac{\mathsf{E}(\Phi,\phi_1 \mathsf{V} \phi_2)}{\mathsf{E}(\Phi,\phi_1,\phi_2) \quad \mathsf{E}(\Phi,\phi_2,\mathsf{X}(\phi_1 \mathsf{V} \phi_2))} \qquad R10\ \vee : \frac{\mathsf{E}(\Phi,\phi_1 \mathsf{U} \phi_2)}{\mathsf{E}(\Phi,\phi_2) \quad \mathsf{E}(\Phi,\phi_1,\mathsf{X}(\phi_1 \mathsf{U} \phi_2))}$$

$$R11\ \langle\Psi\rangle : \frac{\mathsf{E}(\Psi,\mathsf{X}\phi_1,\ldots,\mathsf{X}\phi_n)}{\mathsf{E}(\phi_1,\ldots,\phi_n)} \qquad R12\ \langle\langle\Gamma\rangle\rangle : \frac{\mathsf{E}(\Gamma,\mathsf{X}\phi_1,\ldots,\mathsf{X}\phi_n)}{\mathsf{E}(\phi_1,\ldots,\phi_n)}$$

$\Psi$ is a positive set of action literals, while $\Gamma$ is a negative set.

**Fig. 1.** Tableau rules for GCTL*

has no transitions. $\mathsf{X}$ represents the "next-time operator" and has the usual semantics when the path is not deadlocked. A deadlocked path of form $s$ satisfies $\mathsf{X}\Phi$ if $s$ satisfies $\Phi$. $\phi_1 \mathsf{U} \phi_2$ holds of a path if $\phi_1$ remains true until $\phi_2$ becomes true. The constructor $\mathsf{V}$ may be thought of as a "release operator"; a path satisfies $\phi_1 \mathsf{V} \phi_2$ if $\phi_2$ remains true until $\phi_1$ "releases" the path from the obligation. This operator is the dual of the until operator. The details of the semantics are standard and omitted.

In GCTL* $\mathsf{X}$ is self-dual. Thus, while the application of negation is restricted in this logic we nevertheless have the following.

**Lemma 2.** *Let $\psi$ be a state formula in GCTL*. Then there exists a formula $\mathsf{neg}(\psi)$ such that any state in any TS satisfies $\mathsf{neg}(\psi)$ if and only if it does not satisfy $\psi$.*

Our approach to generating ABTAs from GCTL* formulas uses goal-directed rules to construct tableaux from formulas. These rules operate on "formulas" of the form $E\Phi$ and $A\Phi$, where $\Phi$ is a set of path formulas. Intuitively, these terms are short-hand for $\mathsf{E}(\bigwedge_{\phi\in\Phi} \phi)$ and $\mathsf{A}(\bigvee_{\phi\in\Phi} \phi)$, respectively. We also call a set $\Theta$ of action literals *positive* if it contains some $\theta \in \mathcal{A}_{\mathrm{act}}$. Otherwise it is referred to as *negative*. We use $\Psi, \Psi_1, \ldots$... and $\Gamma, \Gamma_1, \ldots$ to denote positive sets and negative sets respectively.

To construct an ABTA for state formula $\psi$ one first generates the states and transitions. Intuitively states will correspond to state formulas, with the initial state being $\psi$ itself. To generate new states from an existing state $\psi'$, one applies the rules in Figure 1 to $\psi'$ in the order specified. That is, one determines which of R1–12 is applicable to $\psi'$, beginning with R1, by comparing the form of $\psi'$ to the formula appearing in the "goal position" of each rule. The label of the rule then becomes the label of the state, and the subgoals of the rule are then added as states (if necessary), and transitions from $\psi'$ to these states added. Leaves are labeled by the state literals they contain. This procedure is repeated until no new states are added; it is guaranteed to terminate [6].

For notational simplicity we have introduced a new label in Rule R12. Intuitively, if an ABTA state is labeled $\langle\langle\Gamma\rangle\rangle$ then it behaves like $\langle\Gamma\rangle$ for nondeadlocked TS states. For deadlocked states, the state is required to satisfy the single descendant. This operator can be encoded using the other ABTA constructs.

To define the acceptance condition $\mathcal{F}$, suppose $\phi \equiv \phi_1 \mathsf{U} \phi_2 \in q$ and let $F_\phi = \{\, q' \in Q \mid (\phi \notin q' \textit{ and } X\phi \notin q') \textit{ or } \phi_2 \in q' \,\}$. Then $\mathcal{F} = \{\, F_\phi \mid \phi \equiv \phi_1 \mathsf{U} \phi_2 \textit{ and } \exists q \in Q.\phi \in q \,\}$. We now have the following [6].

**Theorem 10.** *Let $\psi$ be a GCTL* formula and let $\mathcal{B}_\psi$ be the BTA obtained by the translation procedure described above. Then the following hold.*

1. *$B_\psi$ is and-restricted.*
2. *Let $\mathcal{T} \equiv \langle S, \rightarrow, L, s_0 \rangle$ be a TS. Then $s_0 \models_{\mathcal{T}} p$ if and only if $\mathcal{T}$ is accepted by $\mathcal{B}_p$.*

In general $B_\psi$ will be exponential in the size of $\psi$. However, if $\psi$ falls within the GCTL fragment of GCTL*, then $B_\psi$ is linear in the size of $\psi$.

We close this section with some comments on and-restrictedness. Our model-checking routine only works on and-restricted ABTAs, which means that the rule-based approach described above for producing model checkers only works if the rules generate and-restricted ABTAs. In practice this means that in rules labeled by $\wedge$, at most one subgoal can be "recursive", i.e. can include the formula identified in the goal. For temporal logics based on CTL* this restriction is not problematic, since the recursive characterizations of standard modalities only involve one "recursive call". For logics such as the mu-calculus in which arbitrary recursive properties may be specified the relevant rules would not satisfy this restriction, and the approach advocated in this paper would not be applicable. (It should be noted, however, that sublogics of the mu-calculus, such as the L2 fragment identified in [17], do fit into our framework.)

## 6 Implementation and Empirical Assessment

To assess our ideas in practice we implemented ABTAs in the CWB-NC verification tool [12]. The procedures we coded (in Standard ML) included the following.

- Basic ABTA manipulation routines (819 lines).
- The ABTA model-checking routine given in Section 3 (631 lines).
- ABTA simplification routines described in Section 4 (654 lines).

The routines made heavy use of existing CWB-NC data structures for manipulating automata. This code is "generic" in the sense that it would be used by any ABTA-based model checker implemented in the CWB-NC.

We also implemented a front-end for GCTL* using the Process Algebra Compiler (PAC) [13], a parser- and semantic-routine generator for the CWB-NC. We used sets of actions as atomic action propositions and included only "true" and "false" as atomic state propositions, with with obvious interpretations. The code for the front-end included 214 lines of yacc and 605 lines of auxiliary code, with most of the latter being devoted to the calculation of acceptance-set information and the implementation of Rule 5 in Figure 1. It should be noted that of this code, approximately 15% is GCTL* specific; the rest could also be used defining e.g. a CTL* model checker.

To study the performance of our implementation we used two existing case studies included in the current distribution of the CWB-NC; in each case our goal was to determine how well our generic ABTA-based model checker for GCTL*performed in comparison with the model checker for the L2 fragment of the mu-calculus that is included in the CWB-NC release. The systems studied included a rendering of the SCSI-2 Bus Protocol in a real-time extension of CCS [7] and a description of the Slow-Scan fault-tolerant communications protocol used by British Rail [11]. In both applications mu-calculus formulas encode key properties of the systems in question. We used the existing models but translated the formulas in question into GCTL*; we then ran our ABTA-based model checker for GCTL* in order to compare its efficiency with the CWB-NC's mu-calculus checker. We also performed a deadlock-freedom check in both logics as well.

The properties included several involving fairness constraints. Emblematic of these is Property 2 in [7], which asserts that any phase in the SCSI-2 protocol eventually ends, provided the initiator in the protocol does not repeatedly issue an `ATN` signal. This property may be encoded in GCTL* as follows

$$\mathsf{AG}(\{\texttt{@begin\_Phase}\} \Rightarrow (\mathsf{F}\{\texttt{@end\_Phase}\} \vee \mathsf{GF}\{\texttt{@obs\_setATN,@obsplace}\}))$$

This formula asserts that along all paths, whenever the action `@begin_Phase` occurs, then either the action `@end_Phase` is performed or at least one of the actions `@obs_setATN` and `@obsplace` occurs infinitely often. (The `@obspace` action is needed for reasons relating to the modeling.) The corresponding mu-calculus used in the case study is given below.

$$\begin{aligned}&\neg(\mu X.\langle\texttt{@begin\_Phase}\rangle(\mu Y.\nu Z.\langle\texttt{@begin\_Phase}\rangle\texttt{tt}\vee\\ &\quad\langle\texttt{@end\_Phase}\rangle X \vee \langle-\{\texttt{@obsplace,@obs\_setATN,@end\_Phase}\}\rangle Z \vee \langle\{\texttt{@obsplace,@obs\_setATN}\}\rangle Y)\\ &\quad\vee\langle-\{\texttt{@begin\_Phase}\}\rangle X)\end{aligned}$$

Fairness constraints in the Slow-Scan example are used to ensure that properties only constrain *non-Zeno* executions, i.e. paths in which clock ticks occur infinitely often.

**Table 1.** SCSI-2 performance data for ABTA model checker. All times are reported in seconds.

| Reference # in [7] | Unsimplified ABTA size | Simplified ABTA size | ABTA Time | Mu-calculus Time |
|---|---|---|---|---|
| 1 | 42 | 24 | 2739.670 | 3423.990 |
| 2 | 54 | 8 | 533.400 | 1022.430 |
| 3 | 12 | 8 | 676.220 | 542.180 |
| 4 | 12 | 8 | 401.300 | 483.470 |
| 5 | 42 | 20 | 410.540 | 943.560 |
| 6 | 57 | 8 | 509.420 | 984.600 |
| NoDeadlock | 7 | 5 | 593.240 | 704.850 |

Tables 1 and 2 give our experimental results. For each of the formulas we record four pieces of data: the size of the ABTA before and after simplication, the running time of the ABTA-based GCTL* model checker, and the running time of the CWB-NC model checker on the equivalent mu-calculus formula. Timing information was collected on a Sun Enterprise E450 with two 336 MHz processors and 2 GB of main memory. Some comments are in order.

- ABTA timing information includes all time needed to convert GCTL* formulas into ABTAs.
- Some of the ABTA state-space reduction in our experiments is due to our encoding of the temporal-logic constructs F and G in terms of U and V. These encodings use the constants tt ("true") and ff ("false"), which constant-propagation then eliminates. Introducing explicit rules for these constructs would lead to smaller initial ABTAs, at the expense of a larger set of rules.
- The paper [7] describes two models, the larger of which contains over 62,000 reachable states, while [11] describes four models, with the largest having almost 12,000 reachable states. In each case we used the largest model.
- The mu-calculus model-checker implements the on-the-fly algorithm given in [5, 17], which runs in $O(|M| \cdot |\phi| \cdot ad(\phi))$, where $|M|$ is the size of the system, $|\phi|$ the size of the formula, and $ad(\phi)$ the alternation depth of $\phi$.
- In the SCSI-2 example, Formulas 2, 5 and 6 involve fairness constraints, with 2 and 6 having virtually the same shape. Formulas 1, 3 and 4 are safety properties, with 3 and 4 having the same shape. Consequently, the minimized automata for 2 and 6 have the same number of states, as do 3 and 4. That 2 and 3 have the same number of states is a coincidence.
- In the Slow-Scan example, Formulas 1, 2, 8 and 9 involve fairness constraints, while the others do not.
- Because the translation procedure in Figure 1 treats A by dualizing it (i.e. converting it into ¬E¬), the ABTA for deadlock-freedom has more states than one might expect.

Based on the figures in the tables, we can draw the following conclusions.

1. *The ABTA model checker dramatically outperforms the mu-calculus checker on formulas involving fairness.* In the cases considered, the factor by which the time required by the latter exceeded that needed by the former ranged from 1.9 (SCSI-2 Property 6) to 5.3 (Slow-Scan Property 9), with the average being 3.1. This behavior is a result of the fact that due to the fairness constraints, the mu-calculus formulas all have alternation-depth 2, and the time-complexity of the mu-calculus model checker is affected by alternation depth. The numbers suggest that interestingly, this theoretical distinction in the algorithms has practical ramifications.
2. *The ABTA model checker also tends to outperform the mu-calculus checker for formulas not involving fairness.* In all but two case (SCSI-2 Property 3, Slow-Scan no-deadlock) the ABTA routine outperforms the mu-calculus routine, with the over-all average improvement factor being 1.6.

**Table 2.** Slow-Scan performance data for ABTA model checker. All times are reported in seconds.

| Name | Reference # in [11] | Unsimplified ABTA size | Simplified ABTA size | ABTA Time | Mu-calculus Time |
|---|---|---|---|---|---|
| *failures-responded* | 1 | 52 | 13 | 2.890 | 13.600 |
| *failures-responded-again* | 2 | 59 | 16 | 144.720 | 471.780 |
| *can-tick* | 3 | 12 | 8 | 205.580 | 328.430 |
| *failures-possible* | 4 | 5 | 4 | 0.020 | 0.080 |
| *failures-possible-again* | 5 | 14 | 9 | 118.790 | 189.380 |
| *no-false-alarms* | 6 | 7 | 5 | 1.670 | 2.760 |
| *no-false-alarms-again* | 7 | 14 | 8 | 139.210 | 221.540 |
| *eventually-silent* | 8 | 92 | 14 | 159.710 | 409.190 |
| *react-on-repair* | 9 | 26 | 10 | 137.630 | 729.550 |
| *no-deadlock* | - | 7 | 5 | 205.930 | 200.220 |

## 7 Related Work

Alternating tree automata are studied extensively as a basis for branching-time model checking in [23], and the work presented in this paper is in much the same spirit. However, ABTAs differ from the automata in [23] in ways that we believe ease their use in practice; we summarize the main ones below.

**Transition relation:** In [23] the authors embed propositional constructs inside the transition relation (this is consistent with traditional alternating automata). In ABTAs propositional constructs are used to label states. This offers advantages when ABTAs are simplified; for example, we may use the traditional notion of bisimulation equivalence to minimize ABTAs, since transitions are given as a relation, not as a logical formula.

**Negation:** The automata in [23] do not use negation in the definition of their (alternating) transition relation; ABTAs do allow the use of a negation operator to label states. This allows the acceptance component of an ABTA to be simpler ("Büchi-like") than the Rabin condition in [23] and also simplifies the model-checking algorithm.

**Algorithm:** Because of our Büchi-like condition and our consideration of and-restricted ABTAs, we are able to adapt the memory-efficient on-the-fly algorithm of [15], which is also time-efficient. The time-efficient algorithm of [23] relies on the construction of strongly-connected components, which our algorithm avoids.

We reiterate that and-restricted alternating automata differ markedly from hesitant alternating automata as introduced in [23]. In particular, and-restricted ABTAs require no definition of "levels of weakness" or classification of states as existential/universal. The price we must pay is that "recursion through $\wedge$" is limited. In practice, however, this restriction does not appear limiting.

Another alternating-tree-automaton-based approach to model checking may be found in [30]. The algorithm relies on the use of games to avoid the construction of the strongly-connected components used in [23]. An implementation related to this routine is described in [31], which also gives preliminary performance comparisons with SPIN [21].

Methods for simplifying Büchi word automata have been presented in [19, 28]. The papers both present different simulation-based techniques for reducing the number of states in such automata while preserving the languages accepted. In addition, [28] shows how acceptance sets for generalized Büchi automata may be reduced using a method similar to ours. Neither paper considers alternating or tree automata.

The mu-calculus [22] has also been proposed as an intermediate language for translation-based model checking [3, 6, 16, 18]. Tool support for this translational-scheme remains problematic, however, owing in part to the complexity of the translation procedures for logics like CTL$^*$. Indeed, our interest in ABTAs began with our use of them as intermediate structures for defining a translation of CTL$^*$ into the mu-calculus [6]. Our performance figures also suggest that the alternation-depth factor in mu-calculus model-checking algorithms has practical effects on model-checker performance: our ABTA model-checker performs much better than the mu-calculus checker on formulas with fairness constraints, which have nontrivial alternation-depth.

## 8 Conclusions and Directions for Future Research

This paper presents a generic approach to building model-checkers that relies on the use of intermediate structures called alternating Büchi tableau automata. These automata, which may be seen as variants of alternating tree-automata, support efficient model checking and simplification routines, and they also admit the definition of abstract proof-rule-based translation procedures for temporal formulas into ABTAs. This eases the task of retargeting a model-checker, since one need only specify the translation into ABTAs of the logic in question. We demonstrated the utility of our ideas by developing a translation-based model checker for a variant of CTL*. Although not described in this paper, we have also developed a translation-based model checker for the L2 fragment of the mu-calculus [17], and its performance compares very favorably with an existing special-purpose algorithm for the same logic [5].

As future work we would like to develop automated support for the generation of tranlators, so that users could directly enter proof rules and high-level specification of acceptance conditions. We are also interested in an efficient model-checking algorithm for all ABTAs, and we would like to investigate compositional techniques for ABTAs based on the partial-model-checking ideas of [4]. Finally, it would be interesting to investigate the adaptation of the simulation-based automaton simplifications presented in [19, 28] to the ABTA setting.

## References


1. *Symposium on Logic in Computer Science (LICS '86)*, Cambridge, Massachusetts, June 1986. IEEE Computer Society Press.
2. R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Fifth Annual Symposium on Logic in Computer Science (LICS '90)*, pages 414–425, Philadelphia, June 1990. IEEE Computer Society Press.
3. H.R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, April 1994.
4. H.R. Andersen. Partial model checking. In *Tenth Annual Symposium on Logic in Computer Science (LICS '95)*, pages 398–407, San Diego, July 1995. IEEE Computer Society Press.
5. G. Bhat and R. Cleaveland. Efficient local model checking for fragments of the modal $\mu$-calculus. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 107–126, Passau, Germany, March 1996. Springer-Verlag.
6. G. Bhat and R. Cleaveland. Efficient model checking via the equational $\mu$-calculus. In *Eleventh Annual Symposium on Logic in Computer Science (LICS '96)*, pages 304–312, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
7. G. Bhat, R. Cleaveland, and G. Luettgen. A practical approach to implementing real-time semantics. *Annals of Software Engineering*, 7:127–155, October 1999. special issue on real-time software engineering.
8. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
9. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
10. E.M. Clarke and J.M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.
11. R. Cleaveland, G. Luettgen, V. Natarajan, and S. Sims. Modeling and verifying distributed systems using priorities: A case study. *Software Concepts and Tools*, 17(2):50–62, 1996.
12. R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag.
13. R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, to appear.
14. R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.
15. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.
16. M. Dam. CTL* and ECTL* as fragments of the modal mu-calculus. *Theoretical Computer Science*, 126(1):77–96, April 1994.

17. E.A. Emerson, C. Jutla, and A.P. Sistla. On model-checking for fragments of $\mu$-calculus. In C. Courcoubetis, editor, *Computer Aided Verification (CAV '93)*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396, Elounda, Greece, June/July 1993. Springer-Verlag.
18. E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Symposium on Logic in Computer Science (LICS '86)* [1], pages 267–278.
19. K. Etessami and G. Holzmann. Optimizing buechi automata. In C. Palamidessi, editor, *CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–169, State College, Pennsylvania, August 2000. Springer-Verlag.
20. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the IFIP Symposium on Protocol Specification, Testing and Verification*, pages 3–18, Warsaw, June 1995. Chapman and Hall.
21. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
22. D. Kozen. Results on the propositional $\mu$-calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.
23. O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the Association for Computing Machinery*, 47(2):312–360, March 2000.
24. K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1+2):134–152, October 1997.
25. R. Mateescu and H. Garavel. XTL: A meta-language and tool for temporal logic model-checking. In T. Margaria and B. Steffen, editors, *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT'98)*, Aalborg, Denmark, July 1998. ISSN 0909-3206.
26. R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.
27. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, Turin, April 1982. Springer-Verlag.
28. F. Somenzi and R. Bloem. Efficient buechi automata from LTL formulae. In E.A. Emerson and A.P. Sistla, editors, *Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 247–263, Chicago, July 2000. Springer-Verlag.
29. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS '86)* [1], pages 332–344.
30. W. Visser and H. Barringer. Practical CTL$^*$ model checking - should SPIN be extended? *Software Tools for Technology Transfer*, 2(4):350–365, April 2000.
31. W. Visser, H. Barringer, D. Fellows, G. Gough, and A. Williams. Efficient CTL$^*$ model checking for analysis of rainbow designs. In H. Li and D. Probst, editors, *International Conference on Correct Hardware Design and Verification Methods (CHARME'97)*, pages 128–145, Montréal, October 1997. IFIP WG 10.5, Chapman and Hall.
32. J. Yang, A. Mok, and Farn Wang. Symbolic model checking for event-driven real-time systems. *ACM Transactions on Programming Languages and Systems*, 19(2):386–412, March 1997.

# A Pseudo-code for ABTA Model Checking

```
DFS2(v = ⟨q, s, i⟩, v' = ⟨q', s', i'⟩) : bool =
  mark v visited by DFS2.
  C_r := {v_r ∈ V | E_R(v, v_r)}.
  if v' ∈ C_r then return TRUE.
  foreach v_r ∈ C_r s.t. v_r not marked FALSE do
     if v_r not marked visited by DFS2 then
        if DFS2(v_r, v') then return TRUE.
  return FALSE.
```

**Fig. 2.** DFS2

```
markAndPropagate (v=⟨q,s,i⟩, val : bool) : bool =
  if not val then return FALSE.
  mark v TRUE.
  foreach v' ∈ Depend(v) do
     remove v' from Depend(v);
     markAndPropagate (v', TRUE).
  return TRUE.
```

**Fig. 3.** markAndPropagate

```
DFS1 (v=⟨q,s,i⟩) : bool =
  if v marked TRUE then return TRUE.
  mark v visited by DFS1.
  c_n := {v' ∈ Q | E_N(v,v')}.
  c_r := {v' ∈ Q | E_R(v,v')}.
  case (ℓ(q)):
     p ∈ 𝒜:
        return (markAndPropagate (v, s ∈ ℓ_S(p)))).
     ¬:
        foreach v_n ∈ c_n do
           return (markAndPropagate (v, not DFS1(v_n))).
     [Θ], ∧:
        foreach v_n ∈ c_n do
           if not DFS1(v_n) then return FALSE.
        if c_r = ∅ then
           return (markAndPropagate (v, TRUE)).
        for the v_r ∈ c_r do
           if v_r marked visited by DFS1 then
              insert v in Depend(v_r).
           else
              if DFS1(v_r) then
                return (markAndPropagate (v, TRUE)).
        if (accepting(v)) then
           return (markAndPropagate (v, DFS2(v,v))).
        return FALSE.
     ∨, ⟨Θ⟩:
        foreach v_n ∈ c_n do
           if DFS1(v_n) then
              return (markAndPropagate (v, TRUE)).
        foreach v_r ∈ c_r do
           if v_r marked visited by DFS1 then
              insert v in Depend(v_r).
           else
              if DFS1(v_r) then
                return (markAndPropagate (v, TRUE))
        if (accepting(v)) then
           return (markAndPropagate (v, DFS2(v,v))).
        return FALSE.
```

**Fig. 4.** DFS1