Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Practical CTL* Model Checking

Should SPIN be Extended?

Willem Visser[1] **and Howard Barringer**[2]

[1] RIACS, NASA Ames Research Center, USA. e-mail: `wvisser@ptolemy.arc.nasa.gov`
[2] Department of Computer Science, Manchester University, UK. e-mail: `howard@cs.man.ac.uk`

The date of receipt and acceptance will be inserted by the editor

Abstract. We describe an efficient CTL* model checking algorithm based on alternating automata and games. A CTL* formula, expressing a correctness property, is first translated to a *hesitant alternating automaton* and then composed with a Kripke structure representing the model to be checked, where after this resulting automaton is then checked for nonemptiness. We introduce the *nonemptiness game* that checks the nonemptiness of a hesitant alternating automaton (HAA). In the same way that alternating automata generalises nondeterministic automata, we show that this game for checking the nonemptiness of HAA, generalises the nested depth-first algorithm used to check the nonemptiness of nondeterministic Büchi automata (used in SPIN).

Key words: model checking – Büchi automata – hesitant alternating automata – games – SPIN

1 Introduction

For nearly two decades now, model checking has been a popular method for analysing the correctness of designs [5,20]. A model checker performs the task of checking whether a correctness property, expressed in a temporal logic, is valid in a model of a design. Its popularity is to a large extent due to the fact that the model checking task can be automated when checking finite state systems.

Temporal logics are often used as property-description languages since they can describe event orderings without having to introduce time explicitly. There are two main kinds of temporal logics: *linear* and *branching* [16]. In linear temporal logics, each moment in time has a unique possible future, while in branching temporal logics, each moment in time has several possible futures.

Efficient model checking algorithms for linear time temporal logic were developed by exploiting the close relationship of the temporal logic and nondeterministic automata on infinite words [28,27]. Essentially for each formula one can construct a nondeterministic Büchi automaton (NBA) that accepts the same input language as the formula. Model checking then reduces to checking the nonemptiness of the product of the automaton for the formula and the Kripke structure being checked. In [2,1] it has been shown that a similar approach can be taken for branching time temporal logic by using *hesitant alternating automata* on infinite trees (HAA). Here we show that practically effective model checking can be done for the branching time logic CTL*, by giving an algorithm for checking the nonemptiness of hesitant alternating automata that is *both* space and time efficient. We show that this nonemptiness check can be reformulated as a 2-player game, which we call the *nonemptiness game*. We develop a novel way, by playing so-called *new games*, of ensuring that results obtained during the nonemptiness game can be reused in later stages to make the algorithm both space and time efficient. We show that when specifying properties in the sublogics LTL and CTL of CTL*, the nonemptiness game can be optimised by reducing the number of new games required. In fact, it is the case that when we combine the rules for playing LTL games and CTL games we can do nonemptiness checking for full CTL*. Furthermore, it is easy to see that the LTL nonemptiness game is a reformulation of the nested depth-first search for testing nonemptiness for NBA. The extra rules required for the CTL nonemptiness games therefore capture the extra expressive power of the branching time fragment of CTL*.

The rest of the paper is structured as follows. Section 2 contains the syntax and semantics of CTL*. Section 3 describes automata-theoretic LTL model checking, including a description of the nested depth-first search used within SPIN to check the nonemptiness of NBA. In Section 4 automata-theoretic CTL* model checking with HAA is described (including a brief description of the translation from formulas to HAA). Section 5 describes the nonemptiness game for HAA as well as the use of new games, whereas in section 6 we give the rules for playing the optimised games for LTL, CTL and CTL*. In section 7 we briefly discuss the integration of our algorithm within the SPIN system and in section 8 we conclude with some general comments on the work presented here.

2 Syntax and Semantics of CTL*

CTL* can express both linear and branching time properties, and is more expressive than the linear time logic LTL [19] and the branching time logic CTL [7]. In fact, both these logics are strict sublogics of CTL*. For technical convenience only *positive* CTL* formulas will be used here, i.e. formulas with negations only applied to atomic propositions. Any CTL* formula can be transformed into a positive form by pushing negations inward as far as possible by using De Morgan's laws and dualities (note, this transformation can at most double the size of the formula). There are two types of formulas in CTL*: formulas whose satisfaction is related to states, *state* formulas, and those whose satisfaction is related to paths, *path* formulas. Let $q \in Props$, where $Props$ is a set of atomic propositions. The syntax of CTL* state (S) and path (P) formulas is then given by the following two BNF rules:

$$\begin{aligned}\mathbf{S} &::= true \mid false \mid q \mid \neg q \mid \mathbf{S} \wedge \mathbf{S} \mid \mathbf{S} \vee \mathbf{S} \mid \\ &\quad A\mathbf{P} \mid E\mathbf{P} \\ \mathbf{P} &::= \mathbf{S} \mid \mathbf{P} \wedge \mathbf{P} \mid \mathbf{P} \vee \mathbf{P} \mid X\mathbf{P} \mid \mathbf{P}\ U\ \mathbf{P} \mid \mathbf{P}\ V\ \mathbf{P}\end{aligned}$$

A ("for all") and E ("there exists") are referred to as path quantifiers and X ("next"), U ("Until") and V ("release") as path operators. The sublogics CTL and LTL are now defined as:

CTL Every occurrence of a path operator is immediately preceded by a path quantifier.

LTL Formulas of the form $A\mathbf{P}$ where the only state subformulas of $\mathbf{P}$ are propositions, negations of propositions, $true$ or $false$.

The semantics of CTL* is defined with respect to a *Kripke* structure $K = (Props, S, R, s_0, L)$, where $Props$ is a set of atomic propositions, S is a set of states, $R \subseteq S \times S$ is a transition relation that must be total (for every $s_i \in S$ there exists a s_j such that $(s_i, s_j) \in R$), $s_0 \in S$ is an initial state and $L : S \rightarrow 2^{Props}$ maps each state to the set of atomic propositions true in that state. For $(s_i, s_j) \in R$, s_j is a *successor* of s_i and s_i a *predecessor* of s_j. The branching degree, i.e. the number of successors, of a state s is denoted by $d(s)$. A path in K is an infinite sequence of states, $\pi = s_0, s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in R$ for $i \geq 0$. The suffix $s_i, s_{i+1}, \ldots$ of π is denoted by π^i. $K, s \models \varphi$ indicates that the state formula φ holds at state s and $K, \pi \models \psi$ indicates that the path formula ψ holds at the path π of the Kripke structure K. $s \models \varphi$ and $\pi \models \psi$ are written when K is clear from the context. The relation $\models$ is inductively defined as:

- $\forall s \in S$, $s \models true$ and $s \not\models false$
- $s \models p$ for $p \in Props$ iff $p \in L(s)$
- $s \models \neg p$ for $p \in Props$ iff $p \notin L(s)$
- $s \models \varphi_1 \wedge \varphi_2$ iff $s \models \varphi_1$ and $s \models \varphi_2$
- $s \models \varphi_1 \vee \varphi_2$ iff $s \models \varphi_1$ or $s \models \varphi_2$
- $s \models A\psi$ iff for every path $\pi = s_0, s_1, \ldots$, with $s_0 = s$, $\pi \models \psi$
- $s \models E\psi$ iff there exists a path $\pi = s_0, s_1, \ldots$, with $s_0 = s$, and $\pi \models \psi$
- $\pi \models \varphi$ for a state formula φ, iff $s_0 \models \varphi$ where $\pi = s_0, s_1, \ldots$
- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$
- $\pi \models \psi_1 \vee \psi_2$ iff $\pi \models \psi_1$ or $\pi \models \psi_2$
- $\pi \models X\psi$ iff $\pi^1 \models \psi$
- $\pi \models \psi_1\ U\ \psi_2$ iff $\exists i \geq 0$ such that $\pi^i \models \psi_2$ and $\forall j, 0 \leq j < i, \pi^j \models \psi_1$
- $\pi \models \psi_1\ V\ \psi_2$ iff $\forall i \geq 0$ such that if $\pi^i \not\models \psi_2$ then $\exists j, 0 \leq j < i, \pi^j \models \psi_1$

A state s satisfies $A\psi$ ($E\psi$) if every path (some path) π from the state s satisfies ψ, while a path satisfies a state formula if the initial state of the path does. $X\psi$ holds of a path when ψ is satisfied in the suffix starting in the next state on the path, whereas $\psi_1\ U\ \psi_2$ holds of a path if ψ_1 holds on the path until ψ_2 becomes true. V is the dual of U since $\neg(\psi_1\ U\ \psi_2) = \neg\psi_1\ V\ \neg\psi_2$ and is referred to as the "release" operator: $\psi_1\ V\ \psi_2$ holds for a path, if ψ_2 remains true until ψ_1 "releases" the path from its obligation. The following well known abbreviations will also be used: $F\varphi = true\ U\ \varphi$ and $G\varphi = false\ V\ \varphi$.

3 Automata-Theoretic LTL Model Checking

For linear time temporal logics, notably LTL, a close relationship with nondeterministic automata has been established [28,27]. Essentially, with each linear time formula, an automaton over infinite words is associated that accepts exactly all the computations that satisfy the formula. Therefore if we consider the Kripke structure to be an automaton as well, call it A_K, with the automaton describing the formula, A_ψ, then model checking can be described as a language containment problem: $\mathcal{L}(A_K) \subseteq \mathcal{L}(A_\psi)$. This can be rewritten as a nonemptiness problem of intersecting automata: $\mathcal{L}(A_K) \cap \overline{\mathcal{L}(A_\psi)} = \emptyset$.

3.1 Nondeterministic Büchi word automata

LTL formulas can express properties on infinite behaviours, therefore automata that accept infinite sequences (words) are required. Since nondeterministic Büchi automata (NBA) can accept infinite sequences they are often used for automata-theoretic LTL model checking [28,27].

A Büchi automaton A is a 5-tuple $(\Sigma, S, \rho, s_0, F)$, where Σ is a finite alphabet, S is a finite set of states, $s_0 \in S$ is the initial state[1], $\rho : S \times \Sigma \rightarrow 2^S$ is a transition function and $F \subseteq S$ is the set of accepting states. Intuitively, $\rho(s, a)$ is the set of states A can move into when it reads symbol a when in state s. Since it can move to a set of states, the Büchi automaton is nondeterministic. If an infinite word, $w = a_0, a_1, \ldots$ over Σ is given as input to A then a *run* of A is a sequence $s_0, s_1, \ldots$ where $s_{i+1} \in \rho(s_i, a_i)$, for all $i \geq 0$ (we also refer to a run as a path of states). If we define $inf(\pi)$ as the set of states that occur infinitely often on the infinite path π, then π is an accepting path *iff* $inf(\pi) \cap F \neq \emptyset$.

A Kripke structure $K = (Props, S, R, s_0, L)$ can be viewed as a Büchi automaton $A_K = (\Sigma, S, \rho, s_0, S)$, where $\Sigma = 2^{Props}$ and $s' \in \rho(s, a)$ *iff* $(s, s') \in R$ and $a = L(s)$. The automaton A_K has as its accepting set all the states in the automaton and therefore any run of the automaton is accepting. Thus, $\mathcal{L}(A_K)$ is the set of computations (possible behaviours) of K.

In [27] it is established that given an LTL formula ψ a Büchi automaton A_ψ can be constructed such that $\mathcal{L}(A_\psi)$ is the set of computations that satisfies formula ψ; the number of states of A_ψ is in the order of $2^{O(|\psi|)}$. Furthermore, for Büchi automata the following holds: $\mathcal{L}(\overline{A_\psi}) = \mathcal{L}(A_{\neg\psi})$. Let's define the product of two Büchi automata $A_0 = (\Sigma_0, S_0, \rho_0, s_0^0, F_0)$ and $A_1 = (\Sigma_1, S_1, \rho_1, s_0^1, F_1)$ to be $A_0 \times A_1 = (\Sigma_0 \times \Sigma_1, S_0 \times S_1, \rho, s_0^0 \times s_0^1, F_0 \times F_1)$, with $\rho((p, q), a) = \rho_0(p, a) \times \rho_1(q, a)$. In general the following does not hold for Büchi automata: $\mathcal{L}(A_1 \times A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$, since this implies that the two automata must go infinitely often and *simultaneously* through accepting states. Here, however, since all the states of A_K are accepting we have: $\mathcal{L}(A_K \times A_{\neg\psi}) = \mathcal{L}(A_K) \cap \mathcal{L}(A_{\neg\psi})$. Automata based LTL model checking can therefore be described by the following three steps:

1. Negate formula ψ and create the NBA $A_{\neg\psi}$.
2. Construct the product automaton $A_{K,\neg\psi} = A_K \times A_{\neg\psi}$.
3. **If** $\mathcal{L}(A_{K,\neg\psi}) \neq \emptyset$ report invalid **else** report valid.

3.2 Nonemptiness Checking

A Büchi automaton accepts some word *iff* there exists an accepting state reachable from the initial state and from itself [27]. It is easy to see that a linear time algorithm exists to find such an accepting state: decompose the state graph of the automaton into maximal strongly connected components (SCCs), which can be done in time linear in the size of the automaton [25]; the automaton is nonempty *iff* an accepting state exists in any of the maximal SCCs. Since checking whether a Büchi automaton accepts some word can be done in time linear in the size of the automaton and an LTL formula ψ can be translated to a Büchi automaton with $O(2^{O(|\psi|)})$ states, this gives model checking complexity $O(|S| \times (2^{O(|\psi|)}))$ where $|S|$ is the number of states in the Kripke structure to be checked.

[1] In the general case there is a set of initial states, but for model checking we only require a single initial state.

```
dfs(state s)
 Add (s,0) to VisitedStates;
 FOR each successor t of s DO
  IF (t,0) not in VisitedStates THEN
    dfs(t)
  END
 END
 IF s is an accepting state THEN
   seed = s; 2dfs(s)
 END
END;

2dfs(state s)
 Add (s,1) to VisitedStates;
 FOR each successor t of s DO
  IF (t,1) not in VisitedStates THEN
    2dfs(t)
  END
  ELSIF t==seed THEN report nonempty END
 END
END
```

Fig. 1. Nested DFS

Courcoubetis et al. [6] show that during the nonemptiness check of a Büchi automaton the computation of SCCs can be avoided. Note that constructing SCCs is not very memory efficient since the states in the SCCs must be stored during the procedure. The idea is to use a nested depth-first search to find accepting states that are reachable from themselves.

Such an algorithm is given in Figure 1. *VisitedStates* is a data-structure, usually a hash table, that keeps track of all states already seen during the search. The algorithm works as follows: when the first search backtracks to an accepting state a second search is started to look for a cycle through this state. In [6] it was stated that the memory requirements of the nested depth-first search would be double that of a single depth-first search, but in [11] it is shown that only two bits need to be added to each state to separate the states stored in *VisitedStates*. Unfortunately, the time might double when all the states of the automaton are reachable in both searches and there are no cycles through accepting states.

Of all model checkers, SPIN [14] is probably the most widely used: a recent estimation puts the number of installations at around 4000 [13] with an even spread among commercial and academic usage. Correctness properties in SPIN are specified by the so-called *never claim*, which is essentially a Büchi automaton expressing unacceptable behaviour (hence the name *never claim*). Checking nonemptiness of the automaton comprising the product of the *never claim* with the model of the system is done with the nested depth-first algorithm shown in Figure 1 [14,15]. The product automaton is also built on-the-fly during the depth-first search for memory efficiency. An interface for doing LTL model checking exists by translating a LTL formula to a *never claim* (Büchi automaton).

4 Automata-Theoretic CTL* Model Checking

The automata-theoretic counterpart for branching time temporal logic is automata over infinite trees [26,21]. For branching time, unlike for linear time, satisfiability and model checking complexity do not coincide; model checking is typically much easier than checking satisfiability. Nondeterministic tree automata cannot compete with this gap, essentially since the translation from formulas to automata can incur an exponential blow-up in size. Therefore, when using nondeterministic tree automata as a basis for model checking the resulting algorithm's time complexity will be exponential in the size of the temporal formula.

Since nondeterministic automata have traditionally been used for automata-theoretic model checking and since for certain branching time temporal logics model checking is linear (e.g. CTL), automata-theoretic techniques have been considered inapplicable to branching time model checking. In [2,1] it is shown that the use of *alternating* automata over infinite trees is the automata-theoretic counterpart of branching time temporal logics that allows efficient model checking.

Alternating automata generalise nondeterministic automata, since they can express both existential and universal choice, whereas nondeterministic automata can only express existential choice. In fact, the name refers to the automaton's ability to *alternate* between existential and universal choice. In order to define the alternating automata of interest here, the following definition is required: for a given set X, let $B^+(X)$ be the set of positive Boolean formulas over X (i.e. boolean formulas built from elements in X using $\wedge$ and $\vee$), where the formulas $true$ and $false$ are also allowed. $Y \subset X$ satisfies $\beta \in B^+(X)$ if β is satisfied when assigning true to the members of Y and false to $X - Y$. For example, the set $\{s_0, s_1\}$ satisfies the formula $(s_0 \vee s_2) \wedge (s_1 \vee s_3)$, but the set $\{s_0, s_2\}$ does not.

Let's explain the difference between nondeterministic and alternating Büchi word automata with the aid of the function $B^+(X)$ (defined above). For a nondeterministic Büchi word automaton $A = (\Sigma, S, \delta, s_0, F)$ the transition function δ maps a state s and an input symbol $a \in \Sigma$ to a set of states indicating the possible nondeterministic choice for the automaton's next state. The function δ can be represented by using $B^+(S)$; for example, $\delta(s, a) = \{s_0, s_1, s_2\}$ can be written as $\delta(s, a) = s_0 \vee s_1 \vee s_2$. When using alternating automata, however, δ can be an arbitrary formula from $B^+(S)$; for example

$$\delta(s, a) = (s_0 \wedge s_1) \vee (s_2 \wedge s_3)$$

meaning that the automaton accepts the word aw (where a is a symbol and w is a word) when it accepts a in state s and accepts w from both states s_0 and s_1 or from both s_2 and s_3. The transition combines therefore both existential choice (disjunction) and universal choice (conjunction).

4.1 Alternating Tree Automata

In order to define automata over infinite trees we need the following basic definitions. A *tree* is a connected directed graph, with one *root* node, ε, and every other node has a unique parent. More formally, a tree τ over $\mathbb{N}$ is a subset of $\mathbb{N}^*$, such that if $x \cdot i \in \tau$, where $x \in \mathbb{N}^*$ and $i \in \mathbb{N}$, then also $x \in \tau$, and for all $0 \leq i' < i$, $x \cdot i' \in \tau$. For every $x \in \tau$ the nodes $x \cdot i$ where $i \in \mathbb{N}$ are called the *successors* of x. The *degree* of a node x, $d(x)$, in a tree τ is the number of successors of x in τ. A *leaf* is a node with no successors. A *path* π of a tree τ is a sequence of nodes starting with ε and for every $x \in \pi$, either x is a leaf or there exists a unique $i \in \mathbb{N}$ such that $x \cdot i \in \pi$. Let $inf(\pi)$ be the set of nodes that occur infinitely often on the infinite path π.

An *alternating tree automaton* is a tuple $(\Sigma, D, S, \delta, s_0, F)$. Here Σ is a finite alphabet, $D \subset \mathbb{N}$ is a finite set of branching-degrees, S is a finite set of states, $s_0 \in S$ is the initial state, F is the acceptance condition (the type of condition depends on the type of alternating automata; two types are discussed below) and $\delta : S \times \Sigma \times D \to B^+(\mathbb{N} \times S)$ is a partial transition function, where $\delta(s, a, k) \in B^+(\{0, \ldots, k-1\} \times S)$ for each $s \in S$, $a \in \Sigma$ and $k \in D$ such that $\delta(s, a, k)$ is defined. A *run* r of an alternating automaton A on a tree T is a tree where the root is labelled by s_0 and every other node is labelled by an element of $\mathbb{N}^* \times S$. Each node of r corresponds to a node of T. A node in r, labelled by (x, s), describes a copy of the automaton that reads the node x of T in the state s. Note that many nodes of r can correspond to the same node of T. The labels of a node and its successors have to satisfy the transition function (see example below). A run is *accepting* if all its infinite paths satisfy the acceptance condition F. For example, the Büchi acceptance condition $F \subseteq S$ will be satisfied on an infinite path π *iff* $inf(\pi) \cap F \neq \emptyset$. Note that we can get finite branches in the tree representing the run when either **true** or **false** is read in the transition function. However in an accepting run, only **true** can be found as leaves, since a path containing **false** is trivially not accepting.

Example: Let's consider the following transition function: $\delta(s_0, a, 2) = ((0, s_1) \vee (1, s_2)) \wedge ((0, s_3) \vee (1, s_1))$. When the automaton is in state s_0, reads input a and the branching degree of the input tree is 2 when a is read (i.e. there are 2 successor states from this state in the input tree) then the automaton will make two copies of itself (due to the $\wedge$ in δ) which could then proceed in different ways. One possibility is that one copy of the automaton proceeds to state s_1 and the next input this copy reads is in direction 0 of the input tree (which could be considered to be the first successor of the state in the tree where a was read), the other copy of the automaton could also go to state s_1 but read its input in direction 1 of the input tree (say in the direction of the second successor of the state in which a was read). In fact there are four possibilities for the automaton to proceed, summarised below:

- one copy proceeds in direction 0 in state s_1 and one copy proceeds in direction 0 in s_3.
- one copy proceeds in direction 0 in state s_1 and one copy proceeds in direction 1 in s_1.
- one copy proceeds in direction 1 in state s_2 and one copy proceeds in direction 0 in s_3.
- one copy proceeds in direction 1 in state s_2 and one copy proceeds in direction 1 in s_1.

Weak alternating automata (WAA), introduced by Muller et al. [17], was one of the first types of alternating automata to be used for reasoning about temporal logic. For example, in [18] WAA are used to explain the complexity of decision procedures for certain temporal logics. More recently, WAA were used to define linear time algorithms for model checking CTL [1]. In [2], Bernholtz et al. defined *bounded alternation* WAA, that also allow space efficient CTL model checking. In fact, it was shown that CTL model checking is in NLOGSPACE in the size of the Kripke structure.

A weak alternating automaton is defined as follows. Firstly, it uses a Büchi acceptance condition, $F \subseteq S$. The set of states of a WAA can be partitioned into disjoint sets S_i, such that each S_i is either an *accepting set*, i.e. $S_i \subseteq F$, or is a *rejecting set*, i.e. $S_i \cap F = \emptyset$. Furthermore, a partial order $\leq$ exists on the collection of S_i sets such that for every $s \in S_i$ and $s' \in S_j$ for which s' occurs in $\delta(s, a, k)$ for some $a \in \Sigma$ and $k \in D$, we have $S_j \leq S_i$. Thus, transitions from an S_i either lead to states in the same S_i or a lower one. An infinite path in the run of a WAA will therefore get trapped within some S_i; if this S_i is accepting then the path satisfies the acceptance condition.

Unfortunately WAA cannot be used for model checking CTL*, since CTL* can define languages that are not weakly definable [22]. A stronger acceptance condition is required for automata corresponding to CTL* formulas. In [1] hesitant alternating tree automata (HAA) are defined that have a more restricted transition structure than WAA, but a more powerful acceptance condition. As with WAA, there exists a partial order between disjoint sets S_i of S. Furthermore, each set S_i is classified either as *transient*, *existential* or *universal*, such that for each S_i, and for all $s \in S_i$, $a \in \Sigma$ and $k \in D$ the following holds:

- if S_i is transient, then $\delta(s, a, k)$ contains no elements from S_i. Examples (assume $S_i = \{s_0\}$): $\delta(s_0, a, 2) = (0, s_1) \wedge (1, s_2)$ and $\delta(s_0, a, 2) = ((0, s_1) \vee (1, s_2)) \wedge (0, s_3)$
- if S_i is existential, then $\delta(s, a, k)$ only contains disjunctively related elements of S_i. Examples (assume $S_i = \{s_0\}$): $\delta(s_0, a, 2) = (0, s_0) \vee (1, s_0)$ and $\delta(s_0, a, 2) = ((0, s_0) \vee (1, s_0)) \wedge (0, s_1)$; but for the case $\delta(s_0, a, 2) = ((0, s_0) \wedge (1, s_0)) \vee (0, s_1)$, S_i is not an existential set, because $(0, s_0)$ and $(1, s_0)$ are related by a conjunction and $s_0 \in S_i$.
- if S_i is universal, then $\delta(s, a, k)$ only contains conjunctively related elements of S_i. Examples (assume $S_i = \{s_0\}$): $\delta(s_0, a, 2) = (0, s_0) \wedge (1, s_0)$ and $\delta(s_0, a, 2) = ((0, s_0) \wedge (1, s_0)) \vee (0, s_1)$; but for the case $\delta(s_0, a, 2) = ((0, s_0) \vee (1, s_0)) \wedge (0, s_1)$, S_i is not a universal set, because $(0, s_0)$ and $(1, s_0)$ are related by a disjunction and $s_0 \in S_i$.

The acceptance condition is a pair of sets of states, (G, B). From the above restricted structure of HAA it follows that an infinite path, ϕ, will either get trapped in an existential or universal set, S_i. The path then satisfies (G, B) iff either S_i is existential and $inf(\phi) \cap G \neq \emptyset$ or is universal and $inf(\phi) \cap B = \emptyset$. Here we also define a subclass of HAA, called *1-HAA*, for which every S_i set contains only one state in the partial order. In [31] it is shown that 1-HAA is the automata-theoretic counterpart of CTL, whereas HAA in general correspond to CTL* formulas.

4.2 Translating CTL* to HAA

Here we will only discuss informally the translation of CTL* formulas into HAA. The interested reader is referred to [1] for a detailed analysis of this translation. First, *maximal* state subformulas of a formula φ, $max(\varphi)$, need to be defined: ψ is a maximal state subformula of φ if it is a proper state subformula and there is no other proper state subformula of φ for which ψ is also a proper state subformula. For example, let φ = AF(Xq V AFAGp), then $max(\varphi) = \{q, AFAGp\}$. Secondly, observe that complementing an HAA $A = (\Sigma, D, Q, \delta, q_0, (G, B))$ is $\overline{A} = (\Sigma, D, Q, \overline{\delta}, q_0, (B, G))$, where $\overline{\delta}$ is defined as switching all the true and false values and the $\wedge$ and $\vee$ symbols. For example, if $\delta(q, a, k) = p \vee (true \wedge g)$, then $\overline{\delta} = p \wedge (false \vee g)$.

The first step in the translation of φ is to build the HAA for all the formulas in $max(\varphi)$. The formula φ is now rewritten, as φ', with all the formulas in $max(\varphi)$ replaced by atomic propositions. The formula φ' now consists of path modalities preceded by a path quantifier (A or E) and only has propositions as state formulas, i.e. linear time formula preceded by an A or E. Let's consider the case where we have $\alpha = E\psi$, where ψ is a linear time formula. Informally, the idea is to build an HAA for $E\psi$ over the alphabet $\Sigma' = 2^{max(\alpha)}$ and then to expand it over the alphabet Σ by using the HAA for the formulas in $max(\alpha)$. For the formula $A\psi$, an HAA for $E\neg\psi$ (with the negation pushed inside to the propositions) is built in the above fashion and then complemented. In the construction of the HAA for $E\psi$ the crucial point is the construction of a nondeterministic Büchi word automaton that accepts all the infinite words recognised by ψ. A simple translation exists from this automaton to the HAA for $E\psi$ (see [1]). Unfortunately, the Büchi word automaton is exponential in the size of the ψ [27,10]. This results in the complete translation from a CTL* formula into an HAA being exponential.

Note that we do not translate the linear time formula into an alternating Büchi word automaton, even though this translation is known to be linear [18]; this is because the reduction to a 1-letter nonemptiness problem, which we will see in the next section is crucial for efficient model checking, is impossible for alternating Büchi word automata, but possible for nondeterministic Büchi word automata [1].

Example: Consider the CTL* formula $\varphi = AFGp$. Since it is of the form $A\psi$ we need to negate and complement the HAA for $EGF\neg p$. Note that we do not need to construct an HAA for the maximal formula $(\neg p)$ since it is already a proposition. The nondeterministic Büchi word automaton for $GF\neg p$ has the following transition relation M:

$$M(q_0, \{\neg p\}) = M(q_1, \{\neg p\}) = q_1$$
$$M(q_0, \{p\}) = M(q_1, \{p\}) = q_0$$

with the accepting set $\{q_1\}$ and the initial state q_0. (We implemented an optimised version of the algorithm given in [10] to create Büchi word automata from linear time formulas.) From this we construct the HAA for $EGF\neg p$:

q	$\delta(q, \emptyset, k)$	$\delta(q, \{p\}, k)$
q_0	$\bigvee_{c=0}^{k-1}(c, q_1)$	$\bigvee_{c=0}^{k-1}(c, q_0)$
q_1	$\bigvee_{c=0}^{k-1}(c, q_1)$	$\bigvee_{c=0}^{k-1}(c, q_0)$

with acceptance condition $(\{q_1\}, \{\})$ and initial state q_0. Since this HAA is already defined over the alphabet $2^{Props} = \{\neg p, p\}$, all that remains is to complement it to get the HAA for φ (call this HAA A_{AFGp}):

q	$\delta(q, \emptyset, k)$	$\delta(q, \{p\}, k)$
q_0	$\bigwedge_{c=0}^{k-1}(c, q_1)$	$\bigwedge_{c=0}^{k-1}(c, q_0)$
q_1	$\bigwedge_{c=0}^{k-1}(c, q_1)$	$\bigwedge_{c=0}^{k-1}(c, q_0)$

with acceptance condition $(\{\}, \{q_1\})$ and initial state q_0.

4.3 Nonemptiness Checking for HAA

Let's first consider the general approach to automata-theoretic branching time model checking. Recall that for linear time temporal logic each Kripke structure may correspond to infinitely many computations. Model checking is therefore reduced to checking inclusion between the set of computations allowed by the Kripke structure and the language of an automata describing the formula (section 3). For branching temporal logic, each Kripke structure corresponds to a single nondeterministic computation. Therefore, model checking is reduced to checking the membership of this computation in the language of the automaton describing the formula [32]. This suggests the following automata-based model checking algorithm. Given a branching temporal formula φ and a Kripke structure K with degrees in D:

1. Construct the alternating automaton for the formula, $A_{D,\varphi}$.

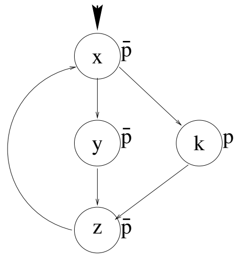

Fig. 2. Kripke structure $K = (\{\{p\}, \{\neg p\}\}, \{x, y, z, k, h\}, R, x, L)$

q	$\delta(q, \emptyset, l)$	$\delta(q, \{p\}, l)$
q_0	$\bigvee_{c=0}^{l-1}(c, q_1) \wedge \bigwedge_{c=0}^{l-1}(c, q_0)$	$\bigwedge_{c=0}^{l-1}(c, q_0)$
q_1	$\bigvee_{c=0}^{l-1}(c, q_1)$	true

Fig. 3. HAA $A_{D,AGEFp} = (\{\emptyset, \{p\}\}, D, \{q_o, q_1\}, \delta, q_0, (\{\}, \{\}))$

2. Construct the product alternating automaton $A_{D,\varphi}^{K} = K \times A_{D,\varphi}$. This automaton simulates a run of $A_{D,\varphi}$ on the tree induced by the Kripke structure K.
3. If the language accepted by $A_{D,\varphi}^{K}$ is nonempty then φ holds for K, otherwise not.

Thus, a nonemptiness check for HAA is required to check CTL* properties in K. The general nonemptiness check for HAA cannot be done efficiently [1]. Fortunately, taking the product with the Kripke structure K, results in a 1-letter HAA over words (i.e. an HAA with $|\Sigma| = 1$ and $D = \{1\}$), for which a nonemptiness check can be done in linear time [1]. Let's now define this product automaton. Let $A_{D,\varphi} = (2^{Props}, D, Q_\varphi, \delta_\varphi, q_0, (G_\varphi, B_\varphi))$ be an HAA which accepts exactly all the D-trees that satisfy φ and let $K = (Props, S, R, s_0, L)$ be a Kripke structure with degrees in D. The product automaton is then an HAA word automaton $A_{D,\varphi}^{K} = (\{a\}, S \times Q_\varphi, \delta, (s_0, q_0), (S \times G_\varphi, S \times B_\varphi))$ where δ is defined as ($succ_R(s)$ is the ordered list of successors of s in K):

- Let $Q \in Q_\varphi$, $s \in S$, $succ_R(s) = (s_0, \ldots, s_{(d(s)-1)})$ and $\delta_\varphi(q, L(s), d(s)) = \alpha$. Then $\delta((s, q), a) = \alpha'$, where α' is obtained from α by replacing each atom (c, q') in α by (s_c, q').

A run of an alternating automaton is a tree; in the sequel we will display this tree as an And-Or tree with each infinite branch truncated when a node is revisited on a branch. Therefore the product automaton will be displayed in this fashion (note we do not show the $\wedge$ and $\vee$ choices when only one successor state exists in the product automaton). For example consider the product automaton of the Kripke structure in Figure 2 and the HAA for the CTL formula AGEFp (Figure 3) given as an And-Or tree in Figure 4.

To illustrate how this product is obtained we show how the run proceeds from the initial state. In the initial state the automaton is in state q_0 and takes as input the label from state x (namely $\neg p$) and its branching degree (2) in the input tree

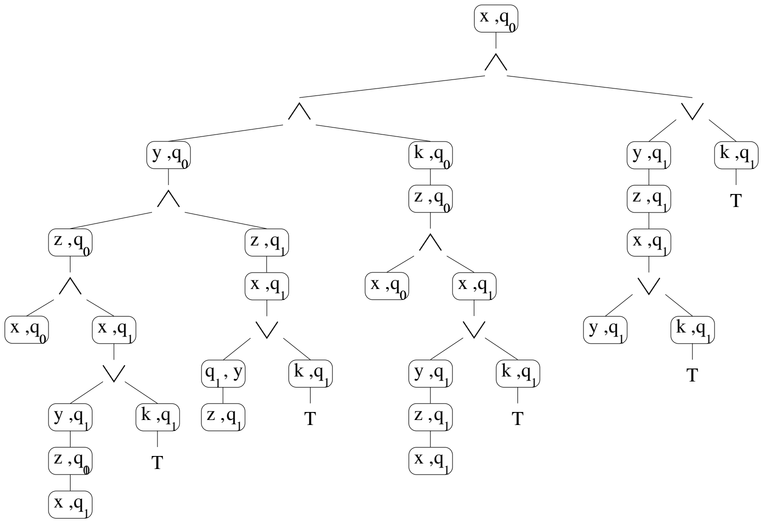

Fig. 4. And-Or tree for the product automaton $K \times A_{D,AGEFp}$.

induced by the Kripke structure K:

$$\delta(q_0, \{\neg p\}, 2) = ((0, q_1) \vee (1, q_1)) \wedge ((0, q_0) \wedge (1, q_0))$$

If we consider y to be the first successor of x and k the second successor of x then we get:

$$\delta(q_0, \{\neg p\}) = ((y, q_1) \vee (k, q_1)) \wedge ((y, q_0) \wedge (k, q_0))$$

which is displayed graphically in Figure 4. Note that all the branches that reach the state (k, q_1) are trivially accepting since a **true** is read in the transition function. All other branches are infinite and their acceptance is determined by the acceptance condition $(\{\}, \{\})$: the infinite branches with a q_0 component are accepting (since q_0 is in a universal set and $q_0 \notin B$, i.e. $inf(\pi) \cap B = \emptyset$, where π is any of the infinite branches with a q_0 component) whereas all the infinite branches with a q_1 component are not accepting (since q_1 is in a existential set and $inf(\pi) \cap G = \emptyset$, where π is any of the infinite branches with a q_1 component). However, since all the infinite branches with a q_1 component can be avoided by picking the other option in the $\vee$-choice and reach a trivial accepting state it follows that the run of the alternating automata is accepting and hence $K, x \models AGEFp$.

In [1] it is shown that the 1-letter nonemptiness problem can be solved in a space efficient and a time efficient fashion. Unfortunately, the algorithm proposed for time efficient nonemptiness checking is incompatible with the one that allows space efficiency. This is because the time efficient algorithm is a global algorithm (product HAA must be kept in memory throughout the procedure) and the space efficient algorithm is a local algorithm (only the part of the HAA being explored is kept in memory). In order to tackle large model checking examples a practical algorithm is required that is both time and space efficient. In the next section we consider the theory of 2-player games as a way of defining such an algorithm for the nonemptiness checking of HAA.

5 Nonemptiness Game

Emerson and Jutla were the first to use game-theory in combination with temporal logic [8]. They used infinite Borel games to show that satisfiability checking for CTL* is in deterministic double exponential time. In [24] Stirling showed how *Ehrenfeucht-Fraïssé* games can be used to capture the expressive power of the extremal fixed point operators of the μ-calculus . To the best of our knowledge, Stirling was the first to use two-player games for model checking [23] when he reformulated the model checking problem for the μ-calculus as a two-player game. Here we show how the nonemptiness check for HAA can be formulated as a 2-player game[2]. We refer to this game as the *nonemptiness game*. We will show that formulating the nonemptiness problem for HAA as a game has two main advantages: (1) The game is simple and can be played without prior knowledge of the automata-theoretic details. (2) Although the game does not improve the worst-case complexity of the nonemptiness check for HAA, it leads to a simple and efficient implementation for checking nonemptiness of an HAA.

The nonemptiness game is defined as a two-player game, in which player 1 will try to show that the HAA is empty whilst player 2 will try to establish that it is nonempty. A *play* of the game is a possibly infinite sequence of *positions*[3] of the form $(q_0, s_0), (q_1, s_1), \ldots$ where each position (q_i, s_i) is a node in an And-Or tree (in our case, the product of the HAA for the formula and the Kripke structure). Which player makes the next move is determined by the structure of the And-Or tree: player 1 (`Brandy`) moves whenever there is an

[2] We do not claim this formulation is new, in fact Gurevich and Harrington [12] showed that the nonemptiness problem for nondeterministic Rabin tree automata can be reduced to a 2-player game, which is closely related to our result.

[3] Positions in the game setting are equivalent to states in the automata setting.

$\wedge$-choice and player 2 (Port) when there is an $\vee$-choice[4]. The players therefore do not take turns as is the case in many standard games. The winner of a play can be determined when either a node that is labelled $true$ (Port wins) or $false$ (Brandy wins) is found in the play or when a position in the current play is revisited.

When a position in a play is revisited it represents the scenario where there is an infinite path in the product HAA and therefore we need to consider the acceptance condition (G, B) to determine which player wins the play. Recall the acceptance condition for an infinite path, π, in an HAA:

1. If π gets trapped in an existential S_i and $inf(\pi) \cap G \neq \emptyset$
2. If π gets trapped in an universal S_i and $inf(\pi) \cap B = \emptyset$

Let's define a set of positions, $infpos(play_\pi)$, to be the positions in the current play, $play_\pi$, that are visited infinitely often on $play_\pi$. Existential S_i only contain disjunctively related elements, thus Port will be making the choices of which move to make in these S_i. Universal S_i, on the other hand, only contain conjunctively related elements, therefore Brandy makes the next move. The definitions of existential and universal sets combined with 1 and 2 above are sufficient to define the winning conditions of a play, summarised in Figure 5.

Since the intentions of the players when making moves are backwards sound, the following rule can be used to combine the results of plays: if Brandy (Port) moves from position s to s' in a play and s' is the start of a winning play for Brandy (Port), then Brandy (Port) also wins from s. A player has a *winning strategy* for a game if the player can win any play of the game from a position, regardless of the opponent's moves.

Theorem 1. *Player 2 (Port) has a winning strategy in the nonemptiness game from the initial state of an HAA iff the language accepted by the HAA is nonempty.*

Proof (sketch): Each play in the game is essentially checking the acceptance of a (possibly) infinite path in the HAA. Since the choices made by the two players in order to find a winning play do not depend on any previous choices, the winning strategy for a player can be considered *history-free* (or memoryless). Therefore, all that is required to prove theorem 1 is to show that if there is an accepting run of an HAA then it will be a history-free run. By [9], and by the fact that in each S_i set of the partition the hesitant acceptance condition is either a Büchi or co-Büchi condition, this is indeed the case, an no memory is required. □

Example: If we play the nonemptiness game on the HAA, $K \times A_{D,AGEFp}$ from Figure 6, then it is clear that whenever Port has a move it can reach the position (k, q_1) which is a winning position for Port (the winning players are shown as labels on the positions in Figure 6). The only interesting plays are those that revisit (x, q_0) and (x, q_1), for example for the left-most play ($play_\pi$), $infpos(play_\pi) = \{(x, q_0), (y, q_0), (z, q_0)\}$, but since B is empty Port wins this play. Port in fact wins every play, regardless of Brandy's moves, and from Theorem 1 it then follows that $K \times A_{D,AGEFp}$ is nonempty, and hence also that $K \models AGEFp$.

Theorem 2. *Given a Kripke structure K and a CTL* formula φ then $K \models \varphi$ iff Player 2 (Port) has a winning strategy for the nonemptiness game on $K \times A_{D,\varphi}$, where $A_{D,\varphi}$ is an HAA such that the language accepted by $A_{D,\varphi}$ is exactly the set of D-trees satisfying φ.*

Proof: Theorem 2 follows directly from Theorem 1 and captures the relationship between the model checking problem and the nonemptiness game. □

From Theorem 2 we can now construct a more effective algorithm for doing CTL* model checking. The first part is to construct the HAA from the CTL* formula and then to play the nonemptiness game on the product of this HAA and the Kripke structure. The construction of the HAA from a CTL* formula is given in section 4.2. Next we will show how to implement the nonemptiness game in an efficient fashion.

5.1 Implementing the Nonemptiness Game

In the previous section it was shown that the moves of the nonemptiness game are determined by the structure of the And-Or tree. We have implemented a depth-first algorithm for finding winning plays in an And-Or tree. An efficient implementation of $infpos$ is obtained by keeping track of the positions in the current play on a stack data-structure, where a new position is pushed every time a move is made and popped whenever a winning play is found from the position. The elements in $infpos$ are therefore all the elements in the stack between the depth where a position is revisited and the current depth (value of the top of stack pointer). The stack is also used to keep track of the other possible moves from a position, but the moves themselves will only be made if the depth-first algorithm requires it later (i.e. after backtracking) in the search for a winning play. For example when looking for a winning play in the initial position of Figure 6 the left-hand choice at the $\wedge$-node is taken and the fact that there is a right-hand choice is recorded in the stack, but it is only explored when it is clear that the left-hand choice returns a winning play for Port. This approach is in general more memory efficient than a breadth-first algorithm where all the choices are explored simultaneously.

Unfortunately, although the algorithm outlined above is memory efficient, it is not time efficient. The reason for this is that "winning" games from a position can be replayed. Considering again the example of Figure 6, it is clear that in the play $(x, q_0), (y, q_0), (z, q_0), (x, q_1)$ there is a winning play for Port in position (x, q_1), but this position arises three more times in other plays and the fact that Port has a win from

[4] The player names reflect when the player moves: Br<u>and</u>y and P<u>or</u>t.

Player 1 (Brandy) Wins	Player 2 (Port) Wins
Play reaches a $false$	Play reaches a $true$
After a move by Port, that revisits a position in the current play, $infpos(play_\pi) \cap G = \emptyset$	After a move by Port, that revisits a position in the current play, $infpos(play_\pi) \cap G \neq \emptyset$
After a move by Brandy, that revisits a position in the current play, $infpos(play_\pi) \cap B \neq \emptyset$	After a move by Brandy, that revisits a position in the current play, $infpos(play_\pi) \cap B = \emptyset$

Fig. 5. Winning Conditions for a Play in the Nonemptiness Game

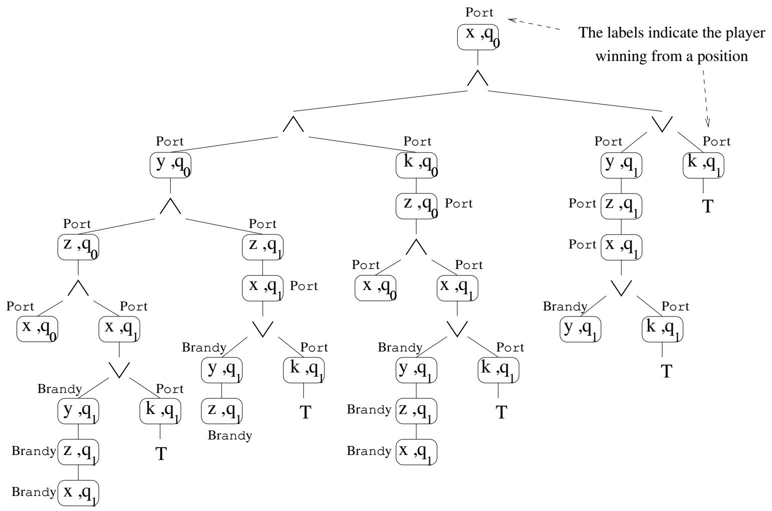

Fig. 6. And-Or tree for the product automaton $K \times A_{D,AGEFp}$

this position will be re-established each time. A *results* store is required to keep track of winning positions so that when they are revisited in different plays then the results can be reused. The problem is however to determine when to store the results, or to put it another way, when a potential winning position is stored to be certain that the winning position is indeed correct. A first consideration might be to store the winning position when all moves from a position have been played (i.e. when the depth-first algorithm backtracks). However, since a play is truncated whenever a position is revisited, it may happen that an incorrect result can be stored when all moves from a position have been made. The problem is that a winning play for a player may now be missed since that play may have been truncated at some point.

5.2 New Games

In order to ensure that the results stored as winning positions are indeed correct, it is proposed that a *new* game is played whenever all the moves from a position, say s, have been played. This new game takes as its initial position s and a new stack and new results store are used. Since a new game uses a new stack and a new store, the intuition is that plays that were previously truncated will be played to completion in the new game, hence ensuring that the correct result is obtained for the initial position of the new game. When a new game is completed (i.e. the winning player from its initial position is found) the stack and results store for the new game are deleted and the result of the new game is stored in the original[5] results store. Whenever a position is visited in the new game it is first checked whether this position is in the original store, since if it is that result can be used. Note that when we refer to the *nonemptiness game* we refer to the initial (first) game together with all the new games that are played in order to determine the winning player for the initial position (of the first game).

New games may have to be played recursively, i.e. whilst playing a new game another new game can be started etc. Therefore, to ensure that new games will not be played infinitely, a new game is not allowed from a position from which a new game is already being played. In fact, when a new game is being played one need not play more new games from positions that are on the current play of any of the previous games (initial game and all new games currently being played). The reason for this restriction is that new games for positions that are on the current play of a previous game will be played later on (precisely, when the positions are backtracked in the previous game). This therefore has the effect of just postponing the new game. Note that the initial position of a new game is by definition also part of the current play of the previous game and therefore this restriction also ensures that new games cannot be played infinitely. Furthermore, when a new game is to be started from a position that is in the acceptance condition (G,B) and a previous new game is already being played for

[5] The store used for the initial game will be referred to as the original store and is never deleted.

this position then a cycle exists and the winning player from this position can be determined by whether it is part of a universal or existential set.

Example: In Figure 7 it is shown how a new game is played when the only move from (z, q_1) has been played and the result of this new game is that (z, q_1) is a winning position for Port. Reusing this result when (z, q_1) is visited again enables Port to win the game from the initial position which is the correct result. No further new games are played for the positions visited in the new game from position (z, q_1), since all these positions are on the current play of the initial game. New games are however played for the positions in the initial game, but in those cases the results are immediately found in the original results store (due to the depth-first nature of the plays).

5.3 Algorithm

An algorithm for playing the nonemptiness game is given in Figure 8. In order to illustrate how the algorithm works we first need to show that some moves made by the players relate only to the structure of the HAA for the formula, whereas other moves depend on the number of successor states from some state in the Kripke structure. For example, if we are in state (x, q_0) in Figure 4 then the next move depends on the following expression (from the transition table of Figure 3):

$$\bigvee_{c=0}^{k-1}(c, q_1) \wedge \bigwedge_{c=0}^{k-1}(c, q_0)$$

The $\wedge$ in the middle of this expression is a move for Brandy, and does not depend on the fact that state x has two successors ($k = 2$). Whereas the moves for $\bigvee_{c=0}^{k-1}(c, q_1)$ (Port moves) and $\bigwedge_{c=0}^{k-1}(c, q_0)$ (Brandy moves) are dependent on the number of successor states of the Kripke structure. We make a distinction between these two types of moves and they are handled by different parts of the algorithm: we call the former AND and OR moves and the ones dependent on the successor states AND_SUC and OR_SUC moves. Essentially this distinction allows the state of the Kripke structure to be hidden in the algorithm of Figure 8: we will only refer to Alt_State which is the state of the HAA for the formula being checked. Alt_State can take on 6 different values: AND, OR, AND_SUC, OR_SUC, TRUE and FALSE.

The algorithm consists of two functions Game and Play that both return a winning player from a specific position. Game, when in the Expand mode, takes care of the lookups in the transition table (using the state of the HAA and the state of the Kripke structure as indexes) of the HAA for the formula to be checked. After an expression is expanded by CreateExpr, Game is called in the Play mode and plays the moves related to the structure of the HAA, i.e. for AND and OR (as defined above). It also checks whether Brandy or Port has a trivial win in the case when respectively a FALSE or TRUE was expanded in the transition table. However, when the next move requires the successor states of the current Kripke state to be evaluated then the Play function is called. Note that when Play is called only one of the two players will be making the moves. The function MoveInK makes a move by generating a successor state in the Kripke structure. If the position reached after this move has not been seen before in the game then Game is called in the Expand mode from this position.

Here follows a brief description of each function being called in Game and Play:

CreateExpr This function does the lookup in the transition table and expands the entry it finds to an And-Or expression or to TRUE or FALSE. Expressions are created in such a fashion that positions from lower S_i sets are to the left of the $\wedge$ or $\vee$. This, combined with the fact that the moves on the left sides of a $\wedge$ or $\vee$ choice are played first (see function Game), means that winning positions are first established for positions in lower S_i sets in the partial order. In terms of the CTL* formulas this has the effect of first establishing the truth-value of all the state-subformulas before the truth-value of the original formula.

MoveInK This function has several purposes, firstly it chooses a successor of the current position, it then checks whether this new position is on the current play, i.e. in the stack, if so it returns a value stating whether a position in B (or G) was found on this cycle, depending on whether it is a move by Brandy or Port that causes the cycle to be closed (B_in_Infpos,....). If a cycle isn't closed by the new position a check is made in the results store to see if this position is winning for a player, if so either Brandy_win or Port_win is returned. Lastly, if the new position is neither on the stack nor in the results store, it is pushed onto the stack and the value new_position is returned. When all the successors have been visited from a position then no_more_moves is returned.

BackTrack Backtrack is called when a winning position for one of the players is found. Backtrack stores this result in the results store and also pops the entry from the stack and returns the state of the HAA so that successors of this (old) state can now be evaluated. The results store is implemented as a hash table for fast access. When Backtrack is called after a new game has finished the result is stored in the original results store (i.e. the one associated with the very first game). This ensures new games will only be played once for every position.

NotPlayedBefore Checks whether a game is already being played from the current position or whether the position is on the current play of any previous game and not in the sets G or B.

PlayNextGame Creates a new results store and new stack for the new game. Note, creating a new store and stack is simply done by incrementing a *Game* counter that would be used as part of the stack and store entries thus avoiding conflict with the entries from other games.

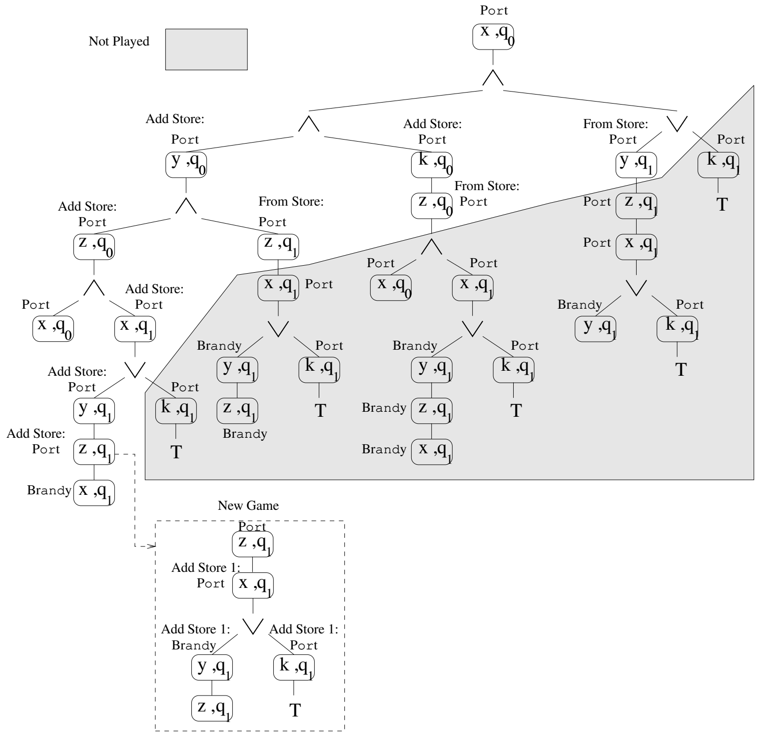

Fig. 7. New Games combined with Results Store

ReturnToPreviousGame Do the reverse of the previous function: clears the new results store and decrements the *Game* counter. Currently the value of the *Game* counter partitions the hash table (results store), thus making it easy to clear the table when a new game is finished. Note the new stack will be empty by definition, due to the `BackTrack` function.

Although the original version of the algorithm was based on a recursive structure (as seen in Figure 8) when we evaluated it on real-world examples [30,31] it became apparent that the overhead (memory in this case) required to keep track of the call stack was too large. We therefore removed as much as possible of the recursion (by eliminating tail recursion) and *inlined* the code for *Play* within *Game*, and as a consequence improved the efficiency by at least an order of magnitude.

5.4 Complexity

Let the number of positions in the product HAA $K \times A_{D,\varphi}$ be n. Since a new game is required for each position and in the worst case each of the n positions needs to be visited for each new game the time complexity of the nonemptiness game is $O(n^2)$. However, due to the depth-first nature of the nonemptiness games and the fact that the original results store is checked for winning positions during a new game, new games tend only to traverse all n positions the first time the game is played and subsequently the results store provides the results. The time complexity of the nonemptiness game is therefore $O(cn)$, where $c \leq n$ and c is the number of new games that will not immediately find a winning position from the results store. Note of course, that c is bound by the number of S_i sets in the HAA for the formula, and ultimately therefore, the length of the formula.

The space complexity of the nonemptiness game is the amount of space required for the stack plus the space required for each of the games' results stores. The space requirement for one results store is linear in the number of positions n, and since we can reuse the space when a game is finished the space complexity is $O(k + n)$, where k is the space required for the stack (in most cases $k \ll n$)

6 Optimised Games

We show that the number of new games played can be reduced by exploiting the structure of the HAA. Firstly we show that checking the nonemptiness of a HAA translated from an LTL formula is similar to the nested depth-first algorithm of section 3.2, next we give the special rules for when to play new games in the CTL case and lastly we consider optimised rules for general CTL* formulas.

6.1 LTL Nonemptiness Games

Here we are interested in CTL* formulas of the form Aφ and Eφ where φ contains no state-subformulas other than

```
Player Play(Alt_State s)
{ while (!Finished) {
  action = MoveInK(s);
  if (action == no_more_moves) Finished = TRUE;
  else {
    switch (s) {
    case AND_SUC:
       switch (action) {
         case new_position:
          if ((result=Game(s,Expand))==Brandy) Finished = TRUE;
           if NotPlayedBefore(s) {
             PlayNextGame();
             if ((result=Game(s,Expand))==Brandy) Finished = TRUE;
             ReturnToPreviousGame(result);
           }
           s = BackTrack(result);
           break;
         case B_in_Infpos    : result = Brandy; Finished = TRUE; break;
         case no_B_in_Infpos : result = Port; break;
         case Brandy_win     : result = Brandy; Finished = TRUE; break;
         case Port_win       : result = Port; break;
       }
       break;
    case OR_SUC:
 // same as case AND_SUC, but with Port substituted for Brandy and G for B.
       break;
} } }
return result;
}

Player Game(Alt_State s, int mode)
{ if (mode == Play)
    if (s == AND)
      if (Game(s.left,Play) == Brandy)
        return Brandy;
      else
        return Game(s.right,Play);
    else if (s == OR)
      if (Game(s.left,Play) == Port)
        return Port;
      else
        return Game(s.right,Play);
    else if (s == TRUE)  return Port;
    else if (s == FALSE) return Brandy;
    else return Play(s);
  if (mode == Expand) {
    s = CreateExpr(s);
    return Game(s,Play);
} }

Initialise:
  1. Generate HAA from formula with initial node of HAA being init.
  2. if (Game(init,Expand) == Port) printf("Formula Satisfied");
     else printf("Formula Invalid");
```

Fig. 8. Algorithm for Nonemptiness Game.

q	$\delta(q,\emptyset,k)$	$\delta(q,\{p\},k)$
q_0	$\bigwedge_{c=0}^{k-1}(c,q_1)$	$\bigwedge_{c=0}^{k-1}(c,q_0)$
q_1	$\bigwedge_{c=0}^{k-1}(c,q_1)$	$\bigwedge_{c=0}^{k-1}(c,q_0)$

Fig. 9. $A_{D,AFGp} = (\{\{p\},\emptyset\}, D, \{q_0,q_1\}, \delta, q_0, (\{\},\{q_1\}))$

$true$, $false$, propositions and their negations (call them linear time formulas). From section 4.2 (and in more detail [1]) we know that the HAA obtained from formulas of the form Eφ only contain $\vee$-choices and those for Aφ only contain $\wedge$-choices.The nonemptiness games for linear time formulas can therefore be considered to be *boring* games for one of the two players, since either all the moves will be made by Port (for Eφ formulas) or by Brandy (for Aφ formulas). Furthermore, when Port makes all the moves the acceptance condition is $(G,\emptyset)$ and when Brandy makes all the moves it is $(\emptyset,B)$ [1]. Therefore from the winning conditions in the nonemptiness game given in Figure 5, if Port (Brandy) moves and G (B) is empty then any position on the current play that is revisited means a win for Brandy (Port). However, if G (or B) is not empty then the fact that the S_i

set is not necessarily singleton means some of the positions in the set can now be in G (B) and other positions not in G (B). For example the HAA translated from the formula AFGp (Figure 9) has an S_i set containing two states (q_0 and q_1) of which only one (q_1) is in the B set. When a nonemptiness game, without any new games, are played for an automaton of the above form, the use of a results store might cause a cycle through a position in G (B) to be missed and consequently a winning position can be labelled incorrectly. A new game is thus required to find a cycle through positions that are either in B or G. This, combined with the fact that only one player moves in a game and that cycles in a play, where no position is in G (B) are trivially labelled, allows us to make the following optimised rule for linear time nonemptiness games:

- During the nonemptiness game for $K \times A_{D,\varphi}$ where φ is a linear time formula, new games need only be played from positions that are either in G or B.

This is the same rule that is used in the nested depth-first search used in the SPIN model checker [15] (see also section 3.2). Here we have presented a justification for this rule in the setting of the nonemptiness game.

6.2 CTL Nonemptiness Games

In [31] it was shown that CTL formulas can be translated to 1-HAA (HAA with singleton S_i sets). Hence, unlike in the linear time case, the introduction of a results store cannot cause a position in G (B) to be labelled incorrectly as a win for the wrong player. Intuitively, when a cycle in a play is found during a CTL nonemptiness game if all the positions in $infpos$ are in G then `Port` wins the play (vice versa for B and `Brandy`); if one of these positions in $infpos$ is revisited in a later play then the result in the store can be reused since the new play will also have a cycle through the positions in $infpos$ (this is not necessarily true in the linear time case). This would seem to indicate that new games need only be played from positions that are neither in G nor B. This rule is however too weak.

In the linear time case we have seen that all the moves in a game are made by the same player. Let's now define S_i sets with all the transitions either consisting of only $\vee$-choices or only $\wedge$-choices to be *one-player* sets. Similarly, a set with transitions consisting of both $\vee$-choices and $\wedge$-choices is called a *two-player* set. In the linear time nonemptiness game all the S_i sets are therefore one-player sets. In the 1-HAA for the CTL formula $AGEFp$, given in Figure 3, the set S_{q_0} (i.e. S_i set containing q_0) is a two-player set, whereas the set S_{q_1} (i.e. S_i set containing q_1) is a one-player set. From the linear time case we know that for a one-player set no new games are required if the positions in the set are not in G nor B. This leads to the following stronger rule:

- During the nonemptiness game for $K \times A_{D,\varphi}$ where φ is a CTL formula, new games need only be played from positions that are neither in G nor in B but are referred to in the transition function of a two-player set.

Again considering the 1-HAA for $AGEFp$ (Figure 3), both S_i sets of the automaton are referred to in the transition function of the two-player set S_{q_0} and therefore new games must be played for all positions visited during the nonemptiness game.

In the CTL nonemptiness game it is unnecessary to play new games for positions in the initial S_i set. The reason is that if we consider the positions in lower S_i sets already to be labelled then the boolean transition function for the states in the initial set reduces to only referring to positions from itself. Therefore, it can be considered to be a one-player set and no new games are required for these positions regardless whether the positions are in G or B. For example in the 1-HAA for $AGEFp$, positions with a q_0 component are in the initial S_{q_0} set and therefore no new games will be played for these positions.

6.3 CTL* Nonemptiness Games

Although in the LTL case all states in the HAA can be considered part of the same S_i set, splitting this set into singular sets where possible can allow more efficient nonemptiness games. In the CTL case we saw that for 1-HAA, when positions from one of the S_i sets are in G or B then no new games are required for these positions. Therefore, in the LTL case if the S_i set of the HAA is divided into smaller S_i sets and it is found that all the positions in G (B) are in singleton S_i sets then no new games are required. For example, consider the HAA for the formula $A(Gp \vee Fq)$ given in Figure 10. If we consider both states of the HAA to be in the same S_i set then new games must be played for positions with a q_1 component since these positions are in B. Whereas if we consider the HAA to have two S_i sets then it is clear that no new games are required.

The different rules for determining when to play new games in the CTL and LTL environments are summarised in Table 1. First, the HAA for the CTL or LTL formula is built and then analysed according to the rules in Table 1. After completing the analysis each state in the HAA will have its *NewGame* flag either set to true or false. When the product with the Kripke structure is taken the product state's *NewGame* flag will be copied from the states of the HAA for the formula. When the depth-first algorithm backtracks from a position the *NewGame* flag is tested to see whether a new game is required from the position, i.e. line 11 of the algorithm in Figure 8 is changed to:

```
if (NotPlayedBefore(s) & (s->NewGame)) {
```

By adding a simple proviso to the rules for playing the nonemptiness game the rules in Table 1 are also sufficient to play nonemptiness game for full CTL*. Here we will only give informal arguments for why this is the case, the interested reader is referred to [31] where it is discussed in detail.

q	$\delta(q, \emptyset, k)$	$\delta(q, \{p\}, k)$	$\delta(q, \{q\}, k)$	$\delta(q, \{p, q\}, k)$
q_0	$\bigwedge_{c=0}^{k-1}(c, q_1)$	$\bigwedge_{c=0}^{k-1}(c, q_0)$	true	true
q_1	$\bigwedge_{c=0}^{k-1}(c, q_1)$	$\bigwedge_{c=0}^{k-1}(c, q_1)$	true	true

Fig. 10. $A_{D,A(Gp \vee Fq)} = (2^{\{p,q\}}, D, \{q_0, q_1\}, \delta, q_0, (\{\}, \{q_1\}))$

LTL		CTL	
1	For all states in HAA, set *NewGame* flag to false.	1	For all states in HAA, set *NewGame* flag to false.
2	Construct *minimal* S_i sets.	2	S_i sets are singleton (i.e. *minimal*).
3	For all states in G (B) and not in a singleton set, set *NewGame* true.	3	For all states not in G or B and referred to in the transition function of a two-player set, set *NewGame* true.
		4	Set *NewGame* for initial state to false.

Table 1. New game rules in nonemptiness Game for CTL and LTL

First observe that a CTL* formula can consist of a boolean combination of CTL and/or LTL formulas or nesting of such formulas. It is easy to see that the HAA for a boolean combination of formulas will adhere to the rules of Table 1, the HAA for a nesting of formulas (e.g. the HAA for AFG(EFp)) will however need additional treatment. In fact we only need the following proviso: when a next move is to be picked by a player, a move that leads to a lower S_i set must be picked first. This has the effect of first playing a game for all subformulas before playing a game for the formula itself. For example for the HAA for AFG(AGEFp) we first play games from the positions of AGEFp, which will be CTL games, before playing the games for the positions of AFGq (where q can be considered the propositional result of playing the game for AGEFp) which will be LTL games. Note that this proviso is implemented in the algorithm given in Figure 8 (see description of CreateExpr function).

An interesting observation is that for a formula that is expressible as both a CTL and LTL formula no new games will be required during the nonemptiness game: HAA for CTL formulas always have singleton S_i sets and HAA for LTL formulas can only contain one-player S_i, therefore from Table 1 it follows no new games will be played. Note that this result also indicates that for Büchi automata for which all cycles are self-loops, no second search will ever be necessary during a nonemptiness check. This result can be used to reduce the number of nested depth-first searches required within SPIN.

7 Implementing the Nonemptiness game in SPIN

In the title we ask "should SPIN be extended?" such that CTL* properties can be checked. This is not just a question of programming effort it is also a more philosophical question: what is the practical worth of CTL* (or CTL for that matter) for property specification during model checking? There is most definitely a large group of model checking practitioners who believe the branching time fragment of CTL* has nothing to offer over LTL; in response some would offer $AG(reset \rightarrow EFinit)$ (which is not expressible in LTL). The algorithm described here however coincides with the best-known practical algorithm for LTL model checking, hence if one never uses the additional expressive power of CTL* over LTL (or HAA over Büchi automata) then there is no potential performance penalty (see below for when this might happen). Next we therefore consider the practical side of implementing the algorithm described here within SPIN.

It is obvious that it would be simple to implement the LTL nonemptiness game in SPIN. However, adding the CTL* nonemptiness games would require some fundamental changes. Firstly, it is worth noting that whereas for the LTL games at most one new game will be required (from the result for the nested depth-first search [11]), in the CTL* case many (nested) games might be required depending on the formula to be checked. Also, in the CTL* case we currently require that the results stores for the new games be deleted when the games terminate[6], which is not necessary for the strictly LTL case (again from [11]). The biggest single change to SPIN would be to replace the *never* claim with a different notation to handle the positive boolean functions used with the HAA. Specifically to handle the case where we have $\wedge$ connectives.

We have implemented a CTL* model checker based on the work presented here [31]. One of the features of this model checker is that it was designed with the view to easily facilitate changing the design formalism to be checked. Currently we check asynchronous hardware designs described in the Rainbow formalism [30]. A potentially easier avenue for integrating CTL* model checking within the SPIN community might therefore be to check PROMELA designs with our current model checker (rather than change SPIN to play the nonemptiness game).

[6] This requirement needs further investigation; we believe it might not be necessary, although it is necessary in our prototype implementation.

8 Concluding Remarks

The main focus of this paper was to introduce an automata-based model checking algorithm for CTL* that can be implemented efficiently in practice.

In [4] Bhat et al. also gives an efficient on-the-fly algorithm for CTL* model checking, however our algorithm is different in two fundamental ways: their algorithm is not based on automata but rather checks the properties according to the structure of the formula, and secondly, they build SCCs to check properties whereas we use the idea of new games. Since their algorithm is based on the structure of the formulas being checked it means that the algorithm can take different amounts of time (and space) to check equivalent formulas on the same Kripke structure. For example, in [3] it is observed that their model checker finds it *easier* to check the formula $A(FG(p \wedge Fq))$ than the equivalent formula $A(FGp \wedge GFq)$. The automata approach do not suffer from this problem; for example when using our model checker these two formulas translate to the same HAA. The automata approach do however use extra overhead for the translation from the temporal logic, but this translation is done up-front before model checking and do not influence the model checking process itself (as seen with the above structural approach). Building SCCs to check properties is not memory efficient, especially not when checking reactive systems since it is often the case that large parts of a reactive systems' state space is contained within the same SCC.

Another related algorithm is that of Verguawen and Lewi for doing on-the-fly CTL model checking [29]. Although this algorithm only handles CTL and is not automata based, they also observe the cases in which extra information need to be gathered to check certain properties. In essence they require extra work (memory and time) in precisely the cases in which our algorithm would play new games.

The algorithm given in this paper has the following interesting characteristic (its complexity can be measured by the number of new games required during the nonemptiness game, see section 5.4): when checking properties that can be expressed in both CTL and LTL the algorithm has the lowest complexity (no new games required), but when moving to properties that are strictly linear or branching the complexity increases. We believe this is somehow an intuitive result.

References

1. O. Bernholtz. *Model Checking for Branching Time Temporal Logics*. PhD thesis, The Technion, Haifa, Israel, June 1995.
2. O. Bernholtz, M. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. In *CAV '94: 6th International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, 1994.
3. G. Bhat and R. Cleaveland. Efficient Local Model Checking for Fragments of the Modal μ-Calculus. In *TACAS '96: 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, 1996.
4. G. Bhat, R. Cleaveland, and O. Grumberg. Efficient On-the-fly Model Checking for CTL*. In *Symposium on Logic in Computer Science*, June 1995.
5. E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of IBM Workshop on Logic of Programs*, pages 52–71. Lecture Notes in Computer Science, 131, 1981.
6. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1:275–288, 1992.
7. E.A. Emerson and J.Y. Halpern. Decision Procedures and Expressiveness in the Temporal Logic of Branching Time. *Journal of Computer and System Sciences*, (30):1–24, 1985.
8. E.A. Emerson and C.S. Jutla. Complexity of Tree Automata and Modal Logics of Programs. In *29th annual IEEE Symposium on Foundations of Computer Science*, 1988.
9. E.A. Emerson and C.S. Jutla. Tree Automata, Mu-Calculus and Determinacy. In *32nd annual IEEE Symposium on Foundations of Computer Science*, 1991.
10. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
11. P. Godefroid and G.J. Holzmann. On the Verification of Temporal Properties. In *Participants Proceedings of the 13-th IFIP Symposium on Protocol Specification, Testing, and Verification*, Liège, Belgium, 25-28 May 1993.
12. Y. Gurevich and L. Harrington. Trees, Automata and Games. In *14th Annual ACM symposium on Theory of Computing*, pages 60–65. ACM Press, 1982.
13. G. Holzmann. Invited Presentation, November 1997. Formal Methods Day, Royal Holloway & Bedford NW College, University of London.
14. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
15. G.J. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth First Search. In Jean-Charles Gregoire, Gerard J. Holzmann, and Doron Peled, editors, *Proceedings of the Second Workshop in the SPIN Verification System*. American Mathematical Society, DIMACS/39, August 1996.
16. L. Lamport. Sometimes is sometimes "not never" — on the Temporal Logic of Programs. *Proceedings 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.
17. D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating Automata, the Weak Monadic Theory of the Tree and its Complexity. In *13th International Colloquium on Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, 1986.
18. D.E. Muller, A. Saoudi, and P.E. Schupp. Weak Alternating Automata give a Simple Explanation of why Temporal and Dynamic Logics are Decidable in Exponential Time. In *Third Symposium on Logic in Computer Science*, pages 422–427, July 1988.
19. A. Pnueli. The Temporal Logic of Programs. In *18th annual IEEE-CS Symposium on Foundations of Computer Science*, pages 46–57, 1977.

20. J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, 1982.
21. M.O. Rabin. Decidability of Second Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.
22. M.O. Rabin. Weakly Definable Relations and Special Automata. In *Proceedings of the Symposium on Mathematical Logic and Foundations of Set Theory*, pages 1–23. North Holland, 1970.
23. C. Stirling. Local model checking games. In *CONCUR '95: 6th International Conference on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, 1995.
24. C. Stirling. Games and model mu-calculus. In *TACAS '96: 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, 1996.
25. Robert E. Tarjan. Depth-First Search and Linear Graph Algorithms. *Society for Industrial and Applied Mathematics*, 1(2):146–160, 1972.
26. M. Vardi and P. Wolper. Automata-theoretic Techniques for Modal Logics of Programs. *Journal of Computer and System Science*, 32(5), 1986.
27. M. Vardi and P. Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115(1), 1994.
28. M.Y. Vardi and P. Wolper. An Automata Theoretic Approach to Automatic Program Verification. In *First Symposium on Logic in Computer Science*, pages 322–331, June 1986.
29. B. Vergauwen and J. Lewi. A Linear Local Model Checking Algorithm for CTL. In *CONCUR '93: 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, 1993.
30. W. Visser, H. Barringer, D. Fellows, G. Gough, and A. Williams. Efficient CTL* Model Checking for the Analysis of Rainbow Designs. Proceedings of CHARME '97, Montreal, October 1997.
31. W.C. Visser. *Efficient CTL* Model Checking using Games and Automata*. PhD thesis, Manchester University, June 1998.
32. P. Wolper. On the Relation of Programs and Computations to Models of Temporal Logic. In *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, 1989.