Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science 149 (2006) 33–49
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Language-Emptiness Checking of Alternating Tree Automata Using Symbolic Reachability Analysis

Kairong Qian Albert Nymeyer

School of Computer Science & Engineering
University of New South Wales
Sydney, Austrlia

Abstract

Alternating tree automata and AND/OR graphs provide elegant formalisms that enable branching-time logics to be verified in linear time. The seminal work of Kupferman et al. [7] showed that 1) branching-time model checking is reducible to the language non-emptiness checking of the product of two alternating automata representing the model and property under verification, and 2) the non-emptiness problem can be solved by performing a search on an AND/OR graph representing this product. Their algorithm, however, can only be implemented in an explicit-state model checker because it needs stacks to detect accept and reject runs. In this paper, we propose a BDD-based approach to check the language non-emptiness of the product automaton. We use a technique called "state recording" from Schuppan and Biere [17] to emulate the stack mechanism from explicit-state model checking. This technique allows us to transform the product automaton into a well-defined AND/OR graph. We develop a BDD-based reachability algorithm to efficiently determine whether a solution graph for the AND/OR graph exists and thereby solve the model-checking problem. While "state recording" increases the size of the state space, the advantage of our approach lies in the memory saving BDDs can offer and the potential it opens up for optimisation of the reachability analysis. We remark that this technique always detects the shortest counter-example.

Keywords: alternating tree automata, language emptiness checking, model checking, AND/OR graph, reachability analysis

[1] Email: kairongq@cse.unsw.edu.au
[2] Email: anymeyer@cse.unsw.edu.au

1 Introduction

Model checking is an automatic verification technique that exhaustively enumerates all states of a model of a system [2,3,19]. This method allows a user to detect property violations in a fully automated manner, and verify the system as a whole. Among the various model checking frameworks, symbolic model checking [9] and the automata-theoretic [19] approaches are prominent success stories and many tools have been built based on these approaches [1,6].

Vardi and Wolper [19] used an automata-theoretic approach on *linear-time temporal logic* (LTL). Coupled with a memory-efficient algorithm, the approach was implemented in the popular tool SPIN [6]. Much more recently, Kupferman et al [7] presented a comprehensive automata-theoretic approach for a branching-time temporal logic. Also in that paper, memory-efficient model-checking algorithms were presented for a number of branching-time temporal logics, including *computation tree logic* (CTL). From an implementation point of view, automata-theoretic algorithms can be readily incorporated into explicit-state model checkers, where each state of the automata is represented explicitly using a block of memory bits. In symbolic model checking, a sets of states can be represented by a symbolic data structure such as a BDD. State-space search using BDDs usually computes all successors, called *image*, of a given set of states and therefore individual path information cannot be distinguished. Hence, the algorithm proposed in [7] cannot be implemented directly using BDDs. Our work in this paper is to devise a method that uses BDDs to verify branching-time logics in an automata-theoretic framework.

While branching-time model checking can be solved efficiently by using the fixed-point characterisation of temporal logic formulae, many state-space search optimisation techniques are only applicable to reachability analysis [5,15,14]. There also has been much work in recent years integrating AI techniques and model checking. Much of this work has involved heuristic search algorithms that have been used to reduce the size of the state space of the model [20,4,13]. These techniques are most effective for a reachability analysis where only a single fixed-point is calculated. For general LTL or CTL model checking, it is still not clear how these techniques can be applied. In recent work, Schuppan and Biere [17] proposed a technique called ‘state-recording’ that transforms LTL model checking into a reachability analysis. In so doing, liveness properties are translated into safety properties. The satisfiability of the liveness properties in the original model and the safety properties in the transformed model are equivalent. One of our goals is to generalise this technique and provide a framework to transform branching-time model checking into a reachability analysis.

In the next section we define alternating tree automata. In Section 4 we describe the automata-theoretic approach, and in particular the concept of 'state recording'. The AND/OR search graph formalism we use is briefly presented in Section 3, together with a fixed-point computation of the solution graphs of the search graph. Our BDD-based automata-theoretic approach is explained in Section 5. The implementation is still very much preliminary, but initial results are shown in Section 6. Finally, in Section 7 we present our conclusions. The key contributions of our work are:

- Reduce a branching-time model-checking problem to a reachability analysis by using alternating tree automata and state recording.
- Devise a BDD-based AND/OR-graph search algorithm.
- Generalise the work of Schuppan and Biere [17] by dealing with any type of property.
- Modify the approach of Kupferman et. al. [7] to work in a symbolic setting.

2 Alternating Tree Automata

The motivation for using an alternating tree automaton (ATA) is that it can be constructed in linear-time for a branching-time logic. Here we briefly describe the formal concepts. For a more comprehensive view the readers should refer to [18,7].

A digraph $T = (V_T, E_T)$ is a tree if every node $x \in V_T$ has exactly one incoming edge except for the root x_0, which has no incoming edges. We denote $(x, y) \in E_T$ a directed edge from x to y and call y a *successor* of x. The *degree* of a node $x \in V_T$ is defined to be the number of successors of x, denoted $d(x)$. A *leaf* is a node without successors. A *path* π of a tree T is a set of nodes $\pi \subseteq V_T$ such that the root $x_0 \in \pi$ and for very node $x \in \pi$, x is either a leaf or there exists a unique node y where $(x, y) \in E_T$. Note that we allow infinite trees and infinite paths. Let Σ be an alphabet and $W : V_T \rightarrow \Sigma$ be a labelling function that maps each node of a tree to a letter in Σ. The pair (T, W) is called a Σ-*labelled* tree. Given a Kripke structure $K = (S, R, s_0)$ where S is a set of states, $R : S \rightarrow S$ is a transition relation and $s_0 \in S$ is an initial state, one can unwind K from s_0 and obtain a possibly infinite (Σ-*labelled*) tree where all the labels come from S. This tree is called the *computation tree* of K.

Let $\mathcal{A} = (\Sigma, Q, \delta, Q_0, F)$ be an automaton where Σ is an alphabet, Q is a *finite* set of states, δ is a transition relation, $Q_0 \subseteq Q$ is an initial state and F the acceptance conditions. In a word automaton, given a state q and input letter σ, the transition relation $\delta(q, \sigma) : Q \times \Sigma \rightarrow Q$ maps the pair (q, σ) to a set $\{q_0, q_1, \ldots, q_k\} \subseteq Q$. The automaton is deterministic if the set is a singleton, and non-deterministic otherwise (assuming the transition is total). A run of a word automaton corresponds to an infinite sequence of states $\rho = q_0 q_1 q_2 \ldots\ldots$ such that $q_0 \in Q_0$ and $\exists\sigma(q_{i+1} \in \delta(q_i, \sigma))$.

In a tree automaton, the transition δ maps each state and input letter to a set $\{B_0, B_1, \ldots, B_k\}$ where $B_i \in \mathcal{P}(Q)$. For example, the transition $\delta(q, \sigma) = \{\{q_0, q_1, q_2\}, \{q_3, q_4\}\}$ causes a given state q of the automaton to change, simultaneously and non-deterministically, to one of the two sets. Hence, unlike word automata, a run of a tree automaton will generate an (infinite) computation tree. In general, the nodes of the tree have varying *branching degrees*. If all degrees of any node is contained in a set $\mathcal{D} \subset \mathbf{N}$, then we call the tree a $\mathcal{D}$-tree. Note, if $\mathcal{D}$ is a singleton, and the element of the singleton is 1, then the tree automaton is simply a word automaton.

2.0.1 An alternating tree automaton

generalises a non-deterministic tree automaton defined above by allowing both universal and existential choices for automata state transitions [10,18]. Consider again a non-deterministic tree automaton with transition $\delta(q, \sigma) = \{\{q_0, q_1, q_2\}, \{q_3, q_4\}\}$. If we express the right-hand side as a Boolean formula, e.g., $\delta(q, \sigma) = q_0 \wedge q_1 \wedge q_2 \vee q_3 \wedge q_4$, we can express both 1) transitions within one group and 2) non-deterministic choice between groups, simultaneously. In fact, this Boolean formula is the characteristic function of the two sets. To accommodate the branching degrees of a tree, we add an extra argument to the transition δ, and rewrite the above transition as $\delta(q, \sigma, 2) = (0, q_0) \wedge (1, q_1) \wedge (2, q_2) \vee (0, q_3) \wedge (1, q_4)$.

Automata-theoretic model checking allows a range of acceptance conditions to be used to express different properties [19]. In this paper, we restrict our discussion to a Büchi acceptance condition. If we have an infinite word automaton $(\Sigma, Q, \delta, Q_0, F)$, then $F \subseteq Q$. If we have some infinite run ρ, then, because Q is finite, some $Q_\rho \subseteq Q$ must appear infinitely often in ρ. A run ρ is a (Büchi) accept run iff $Q_\rho \cap F \neq \emptyset$. Because a run in an ATA is actually a (computation) tree, the acceptance condition is defined over a path in the run. Paths in an ATA correspond to runs in word automata. If all paths are accept paths, the run of the automaton is an accept run.

3 Symbolic AND/OR graphs

AND/OR graphs are commonly used in AI to model problem reduction schemes [11]. To solve a non-trivial problem, one decomposes the problem to a number of (smaller) subproblems. Successfully solving the subproblems will produce a final solution to the original problem according to the decomposing conditions.

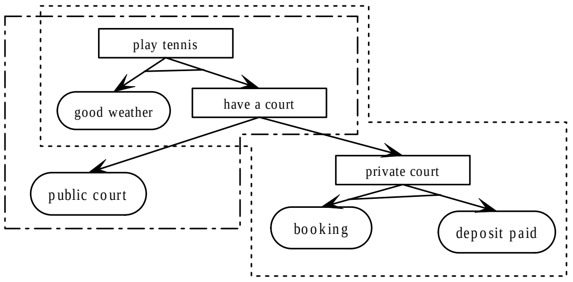

Fig. 1. An AND/OR graph representing the problem of playing tennis

A simple example of such problem reduction is shown in Figure 1.

To play tennis we must have two conjunctive conditions (1) good weather and (2) court available. The problem "good weather" is considered to be an atomic subproblem here so we can treat it as a proposition. The other condition to play tennis is to have a court available and in this case we may have to choose between a public court and a private court. Again, a public court is atomic. A private court can be decomposed into making a booking and paying the deposit. This is essentially similar to the process of natural deduction. Thus, we can say in order to solve "play tennis", we need to solve the problems "good weather" and "have a court" and so on. Eventually we reach the terminal nodes that have the value *true* or *false*.

Formally, an AND/OR graph is a digraph $G = (V, E)$ defined as follows:

- A designated node $n_0 \in V$ is called the *root*.
- There exists a function ζ that maps each node in V to a unique label in $\{\wedge, \vee, \top, \perp\}$
- Nodes with $\zeta(n) = \top$ or $\zeta(n) = \perp$ are called *terminal* nodes and have no outgoing edges to other nodes except themselves.

Given an AND/OR graph $G = (V, E)$, a *solution graph* $G_s = (V_s, E_s)$ is a digraph where:

- $V_s \subseteq V$, $E_s \subseteq E$ and $n_0 \in V_s$
- For each node $n \in V_s$, if $\zeta(n) = \vee$, then only one successor of n is in V_s.
- For each node $n \in V_s$, if $\zeta(n) = \wedge$, then all successors of n are in V_s.
- All finite paths must end with a $\top$-node and all infinite path must have an infinite number of $\top$-nodes as suffix.

The height $\mathcal{H}(G_s)$ of a solution graph G_s is the number of states of the longest prefix before the $\top$-node. In Figure 1 the AND/OR graph has two solution graphs, with heights 2 and 3 respectively, indicated by the dotted and dot-dashed polygons, and they correspond to the two possible solutions of the problem represented by the root node. The existence of the solution graph ideally captures the satisfiability of the property in the model. AND/OR graphs can be coupled with heuristic search algorithms to produce an efficient mechanism to solve the model checking problem [16].

3.1 A symbolic algorithm to detect solution graphs

Algorithms for searching for a solution graph in an AND/OR graph has been studied at great length in AI [11,8]. In general, these algorithms can be categorised as either top-down or bottom-up. Top-down algorithms construct the partial solution graph from the root and use bottom-up propagation to confirm the existence of the solution graph. Bottom-up algorithms construct the solution graph from the terminal nodes and the existence of the solution graph can be determined by checking the reachability of the root. Both these approaches are of course based on explicit-state representations. One of our goals is however to determine a solution graph of an AND/OR graph that is represented with BDDs. When the transition relation is represented by BDDs, it is convenient to compute the predecessor of a given node by computing its *pre-image*. Thus, our approach is in fact bottom-up, i.e. it constructs the graph and checks the reachability of the root in a single run.

Let $G = (S, s_0, R, L)$ be a Kripke structure that induces an AND/OR graph. Here S is set of states, s_0 is the root, $R \subseteq S \times S$ is a transition relation and $L \subseteq S \times \{\wedge, \vee, \top, \bot\}$ is a labelling function. We require L to be total, so all states in S are labelled. We denote the set of states with label $\wedge$ by $S_\wedge$. Similarly for $\vee$, $\top$ and $\bot$. To compute the AND/OR graph symbolically, we need to compute the set of states $S_{sol} \subseteq S$ for all possible solution graphs of G, and then check whether $s_0 \in S_{sol}$. The set S_{sol} is characterised by a fixed-point formula $S_{sol} = \mu Z.(S_\top \cup (EX(Z) \cap S_\vee) \cup (AX(Z) \cap S_\wedge))$.

Theorem 3.1 *S_{sol} contains all states from all solution graphs induced by G.*

Proof. The proof follows from the semantics of a fixed-point and the definition of a solution graph. The right-hand side of the fixed-point formula computes all states of any possible solution graph from the bottom up. It is easy to see the function of the fixed-point is monotonic. The computation starts with the set of states $S_\top$ that are *true* terminals. The component $(EX(Z) \cap S_\vee)$ adds all $\vee$-nodes into S_{sol} and $(AX(Z) \cap S_\wedge)$ adds all $\wedge$-nodes into S_{sol}. The convergence of the fixed-point guarantees all nodes of all possible solution graphs are computed. □

One feature of our work is that the model-checking problem is transformed to a reachability problem. Our approach also generalises the work in [17] and can deal with any type of property. Since the above formula characterises a single fixed-point, the checking of the existence of the solution can be performed on-the-fly: i.e. in every iteration of the calculation we check whether $s_0 \in Z$ twice after $(EX(Z) \cap S_\vee)$ and $(AX(Z) \cap S_\wedge)$. Of course, checking on-the-fly means that we can avoid searching the entire state space before we detect the minimum-height solution graph. In practice, if this optimal graph is a trace, it usually correspond to the shortest counter-example, which is highly desirable for diagnosis purposes of course.

4 Transforming Automata-Theoretic Model Checking to a Reachability Analysis

In this work, we use the automata-theoretic framework to verify CTL formulae. [3] The model-checking problem involves determining whether $M, s_0 \models \varphi$, where $M = (S, R, s_0)$ is a Kripke structure and φ a CTL property. In the automata-theoretic framework proposed in [7], this problem is reduced to a language-emptiness checking problem. We also use this approach, but need to transform the language-emptiness checking to an AND/OR graph reachability analysis.

Let $K = (S, R, s_0, L, AP)$ be a *labelled* Kripke structure where S is a set of states, $R \subseteq S \times S$ is a transition relation, $s_0 \in S$ is an initial state (in general a structure may have a set of initial states) and $L \subseteq S \times AP$ is a labelling function. Let (T_K, W_K) be the computation tree of K, where W_K labels each node of the tree with a letter from 2^{AP}. Let φ be a CTL formula in positive normal form so that negations are pushed inside and placed before atomic propositions by De Morgan's laws. For convenience, we use semantic equivalences during the normalisation of the formulae to reduce the number of operators. Given a formula φ, the closure of ϕ, written $cl(\varphi)$, comprises of formulae that can be defined inductively as follows:

- $\varphi \in cl(\varphi)$, $true \notin cl(\varphi)$ and $false \notin cl(\varphi)$.
- if $\varphi_1 \wedge \varphi_2 \in cl(\varphi)$ or $\varphi_1 \vee \varphi_2 \in cl(\varphi)$, then $\varphi_1 \in cl(\varphi)$ and $\varphi_2 \in cl(\varphi)$.
- if $AX\varphi \in cl(\varphi)$ or $EX\varphi \in cl(\varphi)$, then $\varphi \in cl(\varphi)$.
- if $A(\varphi_1 U \varphi_2) \in cl(\varphi)$, then $\varphi_1, \varphi_2, \varphi_1 \wedge AX(A(\varphi_1 U \varphi_2)) \in cl(\varphi)$.
- if $E(\varphi_1 U \varphi_2) \in cl(\varphi)$, then $\varphi_1, \varphi_2, \varphi_1 \wedge EX(A(\varphi_1 U \varphi_2)) \in cl(\varphi)$.
- if $A(\varphi_1 R \varphi_2) \in cl(\varphi)$, then $\varphi_1, \varphi_2, \varphi_1 \vee AX(A(\varphi_1 R \varphi_2)) \in cl(\varphi)$.
- if $E(\varphi_1 R \varphi_2) \in cl(\varphi)$, then $\varphi_1, \varphi_2, \varphi_1 \vee EX(A(\varphi_1 R \varphi_2)) \in cl(\varphi)$.

The formulae in $cl(\varphi)$ constitute the states of the alternating automaton

[3] Our approach is equally applicable to other branching-time logics such as CTL*, alternation-free μ-Calculus and full μ-Calculus.

$\mathcal{A}_\varphi$ for the CTL formula φ, and the initial state is φ. Following convention, we denote the outermost temporal operator $*$ as a $*$-formula. For example, $A(\mathit{false}\, R\, (E(\mathit{true}\, U\, (\neg p))))$ is a R-formula and EXp is an X-formula. All R-formulae in $cl(\varphi)$ are accept states.

Let $\mathcal{A}_\varphi = (2^{AP}, Q, q_0, \delta_\varphi, \mathcal{L}_\varphi, F)$ be an ATA for a CTL formula φ. We now define the transition relation δ_φ. Note that our construction is different from the one in [7] as we use a labelling function $\mathcal{L}_\varphi$ to label each state of the ATA with a Boolean connective from the set $\{\wedge, \vee, \top, \bot\}$. This allows us to omit connectives in the labels of the transitions. We also use the assumption that if $p \notin \sigma$ then $\neg p \in \sigma$. The transition relation δ_φ and its corresponding labelling function $\mathcal{L}_\varphi$ are defined as follows:

ψ	$\delta_\varphi(\psi, \sigma, k)$	$\mathcal{L}_\varphi(\psi)$
$p \in \sigma$	*true*	$\top$
$p \notin \sigma$	*false*	$\bot$
$\varphi_1 \wedge \varphi_2$	$\{\delta_\varphi(\varphi_1, \sigma, k), \delta_\varphi(\varphi_2, \sigma, k)\}$	$\wedge$
$\varphi_1 \vee \varphi_2$	$\{\delta_\varphi(\varphi_1, \sigma, k), \delta_\varphi(\varphi_2, \sigma, k)\}$	$\vee$
$AX\varphi$	$\{(0, \varphi), (1, \varphi), \ldots, (k-1, \varphi)\}$	$\wedge$
$EX\varphi$	$\{(0, \varphi), (1, \varphi), \ldots, (k-1, \varphi)\}$	$\vee$
$A(\varphi_1 U \varphi_2)$	$\{\delta_\varphi(\varphi_2, \sigma, k), \delta_\varphi(\varphi_1 \wedge AX(\psi), \sigma, k)\}$	$\vee$
$E(\varphi_1 U \varphi_2)$	$\{\delta_\varphi(\varphi_2, \sigma, k), \delta_\varphi(\varphi_1 \wedge EX(\psi), \sigma, k)\}$	$\vee$
$A(\varphi_1 R \varphi_2)$	$\{\delta_\varphi(\varphi_2, \sigma, k), \delta_\varphi(\varphi_1 \vee AX(\psi), \sigma, k)\}$	$\wedge$

Following [7], the product $K \times \mathcal{A}_\varphi$ of the labelled Kripke structure $K = (S, R, s_0, L, AP)$ and property $\mathcal{A}_\varphi = (2^{AP}, Q, q_0, \delta_\varphi, \mathcal{L}_\varphi, F)$ is defined as a 1-letter alternating word automaton $\mathcal{A}_{K,\varphi} = (\{a\}, S \times Q, \langle s_0, q_0\rangle, \delta, \mathcal{L}, S \times F)$. If the property transition $\delta_\varphi(q, L(s), k) = \theta$ and the Kripke transition $R(s) = t_0, \ldots, t_k$ then the transition relation and labelling function of the word automaton are $\delta(\langle s, q\rangle, a) = \theta[(c, q')/\langle t_c, q'\rangle]$ for each c and $\mathcal{L}(\langle s, q\rangle) = \mathcal{L}_\varphi(q)$.

Having defined the product automaton, we now introduce the state-recording mechanism. The idea of the state-recording is to use another set of state variables to copy the state at the beginning of the accept run. The existence of an accept run can be determined by checking whether the current state has been recorded. Since we do not know which state will be at the beginning of an accept cycle, we use a 3-state automaton to guess the beginning of the cycle. We also use a 2-state automaton to indicate whether the state has been recorded. The essence of state-recording is that accept (reject) cycles are flattened, resulting in a product automaton that is a well-defined AND/OR graph. Thus, the model-checking problem is transformed to the problem of determining the existence of a solution graph in an AND/OR graph. The reachability algorithm of Theorem 3.1 can be used to determine the existence of a solution graph and thereby solve the model-checking problem.

Let $S_r = S$ be a set of states that we use to save a copy of a state in K when necessary. Two small Kripke structures K_ς and K_χ are used to determine when the state-recording is necessary, as follows:

	S	s^0	R
$K_\varsigma = (S_\varsigma, s^0_\varsigma, R_\varsigma)$	$\{T, L, R\}$	T	$R_\varsigma(T) = \{L, R\}$, $R_\varsigma(L) = L$, $R_\varsigma(R) = T$
$K_\chi = (S_\chi, s^0_\chi, R_\chi)$	$\{0, 1\}$	0	$R_\chi(0) = 1$, $R_\chi(1) = 1$

The product of $\mathcal{A}_{K,\varphi}, K_\varsigma$ and K_χ after translation is an automaton $\mathcal{A} = (\{a\}, \mathcal{S}, \langle s_0, s_0, q_0, T, 0\rangle, \mathcal{R}, \mathcal{L})$ where $\mathcal{S} = S \times S_r \times Q \times S_\varsigma \times S_\chi$. The transition relation $\mathcal{R}$ is defined as follows:

- $\mathcal{R}(\langle s, s_r, q, s_\varsigma, s_\chi\rangle, a) = R_\varsigma(s_\varsigma) \times \langle s, s_r, q, s_\chi\rangle$ if $s_\varsigma = T$, $s_\chi = 0$ and q is an accept state (guess the beginning of a cycle), or
- $\mathcal{R}(\langle s, s_r, q, L, 0\rangle, a) = \langle s, s, q, L, 1\rangle$ (state recording), or
- $\mathcal{R}(\langle s, s_r, q, s_\varsigma, s_\chi\rangle, a) = R(s) \times \langle s_r\rangle \times \delta(q, a) \times R_\varsigma(s_\varsigma) \times R_\chi(s_\chi)$.

The corresponding labelling function $\mathcal{L}$ is defined as:

- $\mathcal{L}(\langle s, s_r, q, T, 0\rangle) = \vee$ if q is an accept state, or
- $\mathcal{L}(\langle s, s, q, L, 1\rangle) = \top$ if q is an accept state, or
- $\mathcal{L}(\langle s, s_r, q, s_\varsigma, s_\chi\rangle) = \mathcal{L}_\varphi(q)$.

The labelling of states in $\mathcal{A}$ follows the labelling of states in $\mathcal{A}_{K,\varphi}$ except that when we need to guess the beginning of an accept cycle or we have found an accept cycle.

Lemma 4.1 *The graph $G_\mathcal{A}$ induced by $\mathcal{A}$ is a well-defined* AND/OR *graph.*

Proof. It follows directly from the definition of transition relation that $\mathcal{A}_\varphi$ is a well-defined AND/OR graph. The problem is that the product $\mathcal{A}_{K,\varphi}$ will not be well-defined due to the presence of accept (and reject) runs. When the state-recording automata K_ς and K_χ are used, the accept (reject) runs are in essence flattened, changing from cycles to finite traces. The beginning of an accept cycle in $\mathcal{A}$ is a $\vee$-node and another copy, which indicates the existence of the cycle, is a $\top$-node. Thus, the product $\mathcal{A}$ induces a well-defined AND/OR

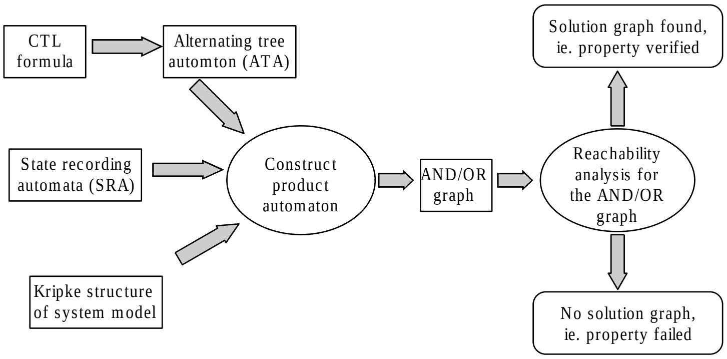
CTL formula
Alternating tree automton (ATA)
State recording automata (SRA)
Kripke structure of system model
Construct product automaton
AND/OR graph
Reachability analysis for the AND/OR graph
Solution graph found, ie. property verified
No solution graph, ie. property failed

Fig. 2. Overview of the automata-theoretic approach

graph. □

Theorem 4.2 *$G_{\mathcal{A}}$ has a solution graph iff the language of $\mathcal{A}_{K,\varphi}$ is non-empty, i.e $K \models \varphi$.*

Proof. The key to the proof is that our state-recording automata are capable of catching all possible accept (reject) cycles. Since we do not know in advance which state will be at the beginning of a cycle, we use in K_ς a non-deterministic transition $R_\varsigma(T) = \{L, R\}$ for every accept (reject) state to guess the starting point. Using our construction rules, this state is labelled as a $\vee$-node. This consideration ensures every possible accept (reject) cycle will be caught. Hence, the existence of a solution graph in $G_{\mathcal{A}}$ confirms non-emptiness of the language of $\mathcal{A}_{K,\varphi}$ and therefore $K \models \varphi$. □

5 Our approach

In essence, our approach, illustrated in Figure 2, uses BDD-based algorithms in an automata-theoretic framework to solve the language non-emptiness problem for branching-time logic. The process begins with the model, represented by a Kripke structure, and the property to be verified, expressed as a CTL formula. Unlike conventional model checking, we also require a state-recording automaton as input. In the first step, the CTL formula is translated to a (weak) alternating tree automata. Then a product automaton for the three inputs is constructed and an AND/OR graph is generated. To determine whether the AND/OR graph has a solution graph, we use a special reachability analysis algorithm. If this algorithm finds a solution graph, the property is verified and the solution graph is a witness of the property. Otherwise, the property is violated. To determine a counter-example, one needs to determine a witness of the negation of the property. The reachability analysis in the AND/OR

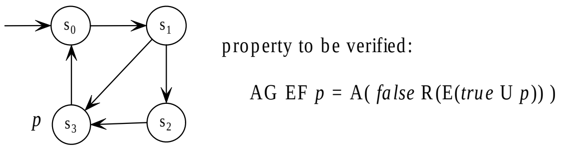

Fig. 3. A Kripke structure and CTL property

graph uses BDDs. We encode the system model, property automaton and state-recording automaton as BDDs, and compute the product automaton and AND/OR graph as BDDs.

5.0.1 *From an ATA to* **AND/OR** *graphs using BDDs*

The key to automata-theoretic branching-time model checking is to determine the language non-emptiness of the product alternating automaton. Kupferman et. al. [7] reduce this problem to a 1-letter automaton non-emptiness checking. While their algorithm uses AND/OR graphs as underlying data structures, the run of the 1-letter automaton does not correspond to an AND/OR graph as we defined above. The serious problem concerns cycle detection. When their algorithm first encounters an accept (or reject) state, it may label the state a $\wedge$-node or $\vee$-node. If subsequently a cycle that goes back to an accept (or reject) state is found, then the same node may be labelled $\top$ or $\bot$. If this occurs, the state does not have a unique label. In a explicit-state model checker, this problem can be avoided by using a stack to memorise the accept cycle and if the cycle does exist, the algorithm just relabels the state. This does not work in a BDD-based model checker because states are represented implicitly and path information cannot be recorded by using a structure like a stack.

5.0.2 *Example*

Consider the Kripke structure and CTL property in Figure 3. The structure has 4 states and s_3 is labelled by the only atomic proposition p (assume $\overline{p}$ labels all other states). The property is obviously true in this model.

To use the algorithm in [7] to verify the property, we first construct the alternating automaton of $AG(EFp)$ as follows:

q	$\delta(q, \{p\}, k)$	$\delta(q, \emptyset, k)$
$AG(EFp)$	$EFp \wedge \bigwedge_{c=0}^{k-1}(c, AG(EFp))$	$EFp \wedge \bigwedge_{c=0}^{k-1}(c, AG(EFp))$
EFp	*true*	$\bigvee_{c=0}^{k-1}(c, EFp)$

where q is the state of the property automaton, the state $AG(EFp)$ is an accept state, and the other state is a reject state. The accept state of the product automaton is determined by the property automaton. Note that if a transition has the value *true*, this means the path leading to that state does not have a successor in a run of the tree automaton. This corresponds to $\top$-node in AND/OR graph. If the transition has the value *false*, the run can never be accepted, i.e. the solution graph can never have a $\perp$-node.

We omit the formal description here of the product automaton and instead depict in Figure 4 the runs that the algorithm in [7] uses to check the non-emptiness of the language. Each node in the graph is a state of the product automaton. The initial state of this automaton contains both the initial states from the Kripke structure and the property automaton. The algorithm begins with the initial state #1, which states that $AG(EFp)$ is true at state s_0. This means that EFp must be true at s_0, which results in node #2, <u>and</u> $AG(EFp)$ must be true at all successors of s_0, namely s_1, which results in node #3. Node #1 is hence a $\wedge$-node. Other states are expanded in a similar manner. The transition *true* results in node #7, which is a $\top$-node. At this point we need to back-propagate the label according to the definition of the solution graph of an AND/OR graph. Since node #4 is a $\vee$-node and one of its children is a $\top$-node, node #4 will be labelled as a $\top$-node as well. The language is non-empty iff the initial state (#1) is labelled a $\top$-node. The problematic state in this example is node #6 because one of its successors loops back to a state that was expanded before. In this situation, the label of node #1 will be recursively dependant on itself (because of the cycle of node #1, #3 and #6). This problem occurs when an automaton has an accept or reject run. In fact, it is easy to see the cycle here is an accept cycle as #1 is an accept state. To avoid this problem, if during the back-propagation of the labels a successor of a node leads to an accept cycle, we consider the node has a $\top$-node as successor. Thus, node #6 is a $\top$-node as #7 is a $\top$-node and node #1 leads to an accept cycle. Alternatively, if a successor of a node leads to a reject cycle, we consider the node has a $\perp$-node as successor. It is easy to see #1 will eventually be labelled as a $\top$-node and the language of the product automaton is not empty.

In an explicit-state model checker, the non-emptiness algorithm above detects accept and reject cycles by maintaining two stacks. Our aim is to use BDD-based algorithms to check for language non-emptiness, and to represent the model under verification and property using BDDs. In general, BDD-based state-space search does not use a stack to record which states have been visited, so the above approach cannot be used directly. Instead, state-recording has been used. The key idea is to use another set of state variables to record the state that possibly lies in a cycle. The existence of the cycle

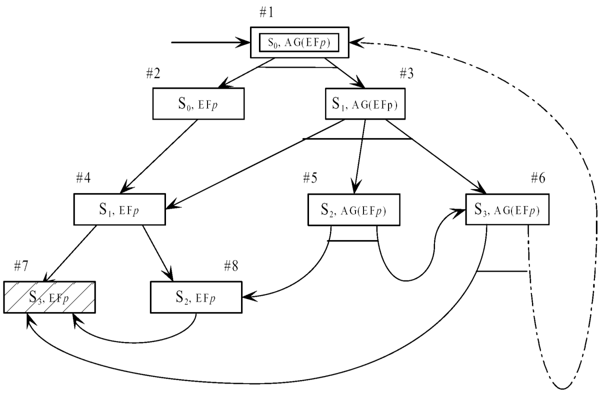

Fig. 4. Automata-theoretic CTL model checking

can be determined by checking whether the value of original state variable is equal to the recorded variable. In effect, state-recording emulates a stack, and cycles can be detected by reachability algorithms.

We use this approach to detect both accept cycles and reject cycles. Figure 5 depicts the search graph that uses this technique. In Figure 4 the state of the product automaton has two components, namely a state of the Kripke structure and a state of the property automaton. After recording the state, the state of the product automaton has 5 components: namely $(s, t, \varphi, save, saved)$, where s is the state of the Kripke structure, t is the state that copies s when necessary, φ is still the state of property automaton, and finally *save* and *saved* are two variables that are used to determine when to record a state and whether the state has already been recorded. To detect an accept cycle, it is sufficient to record the state at an accept state. For example, in Figure 5, node #1 is an accept state and hence *save* will non-deterministically split into two branches. The value for *save* becomes L in #2 and R in #3. If $save = L$, we will copy s to t in the next transition and set the flag *saved* to be 1. Note that the non-deterministic branch leaves open the question whether that state is the beginning of an accept cycle, so the label of this 'split' node is $\vee$. While the graphs in Figure 4 and Figure 5 are similar, the latter contains more nodes. In particular, the shaded node #11 suggests that there is an accept cycle. Node #11 turns out to be a $\top$-node.

By applying state-recording, the product automaton becomes an AND/OR graph in which the label on each state will only depend on its own position. Using our symbolic reachability algorithm, we can determine whether there exists a solution graph and thereby determine the emptiness of the language of the product automaton.

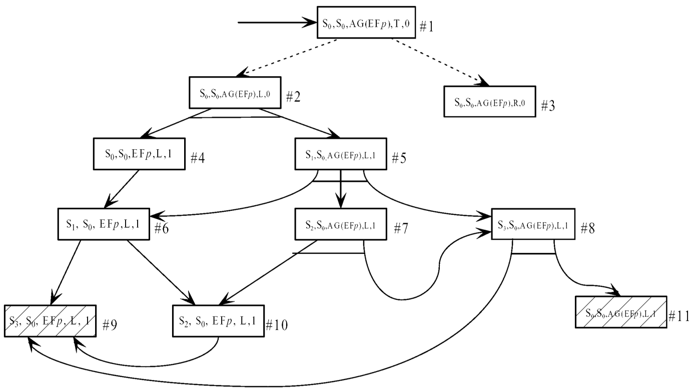

Fig. 5. From an automaton to a well-defined AND/OR graph

6 Implementation

We have implemented this approach in the symbolic model checker NuSMV [1]. The reachability algorithm of the fixed-point formula has been implemented in a separate module and uses both *weak* pre-image and *strong* pre-image routines from NuSMV for the temporal operators EX and AX. We employ most of the features of BDD-based model checking, such as input variable ordering and a partitioned transition relation.

The model under verification is expressed using the SMV input language. We require the input model to our algorithm to be a translated automaton that induces an AND/OR graph. This requires a source-to-source translation from the original model and property to a new input model. To produce the experimental results below we needed to do this translation by hand, but we are developing a tool that can perform this automatically.

The model we experimented on is a *gigamax cache consistency protocol.* The property we verified is $AG(EFp_0.readable)$, which states that no matter what state the system is in, there is a future state from which process p_0 is readable. The property can be verified by both our automata-based approach and the CTL model checking algorithm based on a fixed-point characterisation. A few preliminary results are shown in the following table.

algorithm	# BDD vars	run-time (s)	# BDD nodes	iterations
our approach	188	1.370	425748	4
standard	88	0.250	65947	11

As we need to use a set of variables to save a copy of the original states, our approach uses more than twice the number of Boolean variables than the standard algorithm. Specifically, we use another 88 variables to copy the state, and 12 variables to encode the property automaton as well as the state-recording automata. The net result is a longer run-time and more BDD nodes (and hence memory). The last column shows the number of iterations of the fixed-point calculation. This is a measure of how many times the algorithm calls the pre-image (or image) computation routine, which is normally an expensive operation in symbolic model checking. In this example we call only 4 pre-image operations, which is substantially lower than the number called by the standard algorithm.

7 Conclusion and Future Work

In this work we have in essence transformed a branching-time logic problem into a reachability problem on AND/OR graphs and placed it in a symbolic setting. The big advantage of this transformation from our point of view is that we are now in a position to apply symbolic heuristic-search algorithms during the reachability analysis to reduce the size of the state space. We already have extensive experience with heuristic search [13,12] and feel that the combination of symbolic and heuristic search techniques offers much potential. This work, together with the work of Kupferman et. al. [7], has highlighted the need for really efficient (heuristic) search algorithms to solve what is a quint-essentially AI problem representation, AND/OR graphs.

At one level, our approach will lead to worse performance because of the extra load of the state-recording. Note that this is consistent with Schuppan and Biere's results in [17]. However, in the longer term, the advantage is that, having transformed branching-time model checking to a reachability analysis, an arsenal of optimisation techniques can be applied to improve performance. While we do not explore these techniques in this paper, we will in the near future apply symbolic heuristic search to the reachability analysis of branching-time properties.

While we have focused on CTL model checking, the approach used in this paper is applicable to other branching-time logics such as CTL* and μ-Calculus. In our algorithms we have used Büchi acceptance conditions. In fact, the automata constructed from CTL satisfy 'weak' acceptance conditions and hence these automata are often referred to as weak alternating automata (WAA). In their automata-theoretic approach, Kupferman et. al. [7] construct *hesitant alternating automata* (HAA) for CTL* and μ-Calculus. We have also applied HAA in our symbolic setting to these logics and will report on this work at a later date.

References

[1] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proc. of the 11th Int. Conf. on Computer Aided Verification, CAV'99, Trento, Italy*, volume 1633 of *LNCS*, pages 495–499. Springer-Verlag, 1999.

[2] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[3] E. M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.

[4] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF–SPIN. In *Proc. of Model Checking Software, 8th Int. SPIN Workshop, Toronto, Canada*, volume 2057 of *LNCS*, pages 57–79. Springer-Verlag, 2001.

[5] Ranan Fraer, Gila Kamhi, Barukh Ziv, Moshe Y. Vardi, and Limor Fix. Prioritized traversal: Efficient reachability analysis for verification and falsification. In *Proc. of 12th Int. Conf. on Computer Aided Verification, CAV'00, Chicago, IL, USA*, volume 1855 of *LNCS*, pages 389–402. Springer-Verlag, 2000.

[6] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[7] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, 2000.

[8] A. Mahanti and A. Bagchi. AND/OR graph heuristic search methods. *Journal of the Association fro Computing Machinery*, 32(1):28–51, January 1985.

[9] K.L. McMillan. *Symbolic model checking.* Kluwer Academic Publishers, Boston, MA, 1993.

[10] David E. Muller and Paul E. Schupp. Alternating automata on infinite trees. *Theor. Comput. Sci.*, 54:267–276, 1987.

[11] N. J. Nilsson. *Principles of artificial intelligence.* Morgan Kaufmann Publishers Inc., 1980.

[12] K. Qian and A. Nymeyer. Abstraction-based model checking using heuristical refinement. In *Proc. of the 2nd Int. Symp. on Automated Technology for Verification and Analysis, ATVA'04*, LNCS, pages 165–178. Springer-Verlag, 2004.

[13] K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Proc. of the 10th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'04, Barcelona, Spain,*, volume 2988 of *LNCS*, pages 497–511. Springer-Verlag, 2004.

[14] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proc. of the 1995 IEEE/ACM international conference on Computer-aided design, ICCAD '95, San Jose, California, USA*, pages 154–158. IEEE Computer Society, 1995.

[15] K. Ravi and F. Somenzi. Hints to accelerate symbolic traversal. In *Proc. of Correct Hardware Design and Verification Methods, CHARME'99, Bad Herrenalp, Germany*, volume 1703 of *LNCS*, pages 81–92. Springer-Verlag, 1999.

[16] Antonella Santone. Heuristic search + local model checking in selective mu-calculus. *IEEE Transactions on Software Engineering*, 29(6):510–523, 2003.

[17] Viktor Schuppan and Armin Biere. Efficient reduction of finite state model checking to reachability analysis. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 5(2–3):185–204, 2004.

[18] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 133–192. MIT Press, 1990.

[19] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proc. of the Symp. on Logic in Computer Science, LICS'86, Cambridge, Massachusetts*, pages 332–344. IEEE, 1986.

[20] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Proc. of the 35th Conf. on Design Automation, DAC'98, Moscone center, San Francico, CA, USA*, pages 599–604. ACM Press, 1998.