Truly On-The-Fly LTL Model Checking

Moritz Hammer, Alexander Knapp, Stephan Merz

Truly On-The-Fly LTL Model Checking

Moritz Hammer[1], Alexander Knapp[1], and Stephan Merz[2]

[1] Institut für Informatik, Ludwig-Maximilians-Universität München
{Moritz.Hammer,Alexander.Knapp}@pst.ifi.lmu.de
[2] INRIA Lorraine, LORIA, Nancy
Stephan.Merz@loria.fr

Abstract. We propose a novel algorithm for automata-based LTL model checking that interleaves the construction of the generalized Büchi automaton for the negation of the formula and the emptiness check. Our algorithm first converts the LTL formula into a linear weak alternating automaton; configurations of the alternating automaton correspond to the locations of a generalized Büchi automaton, and a variant of Tarjan's algorithm is used to decide the existence of an accepting run of the product of the transition system and the automaton. Because we avoid an explicit construction of the Büchi automaton, our approach can yield significant improvements in runtime and memory, for large LTL formulas. The algorithm has been implemented within the SPIN model checker, and we present experimental results for some benchmark examples.

1 Introduction

The automata-based approach to linear-time temporal logic (LTL) model checking reduces the problem of deciding whether a formula φ holds of a transition system $\mathcal{T}$ into two subproblems: first, one constructs an automaton $\mathcal{A}_{\neg\varphi}$ that accepts precisely the models of $\neg\varphi$. Second, one uses graph-theoretical algorithms to decide whether the product of $\mathcal{T}$ and $\mathcal{A}_{\neg\varphi}$ admits an accepting run; this is the case if and only if φ does not hold of $\mathcal{T}$. On-the-fly algorithms [2] avoid an explicit construction of the product and are commonly used to decide the second problem. However, the construction of a nondeterministic Büchi (or generalized Büchi) automaton $\mathcal{A}_{\neg\varphi}$ is already of complexity exponential in the length of φ, and several algorithms have been suggested [3,4,5,7,18,20] that improve on the classical method for computing Büchi automata [9]. Still, there are applications, for example when verifying liveness properties over predicate abstractions [13], where the construction of $\mathcal{A}_{\neg\varphi}$ takes a significant fraction of the overall verification time. The relative cost of computing $\mathcal{A}_{\neg\varphi}$ is particularly high when φ does not hold of $\mathcal{T}$, because acceptance cycles are often found rather quickly when they exist.

In this paper we suggest an algorithm for LTL model checking that interleaves the construction of (a structure equivalent to) the automaton and the test for non-emptiness. Technically, the input to our algorithm is a transition system $\mathcal{T}$ and a linear weak alternating automaton (LWAA, alternatively known as a very weak alternating automaton) corresponding to $\neg\varphi$. The size of the LWAA is linear in the length of the LTL formula, and the time for its generation is insignificant. It can be considered as a symbolic representation of the corresponding generalized Büchi automaton (GBA). LWAA have also

been employed as an intermediate format in the algorithms suggested by Gastin and Oddoux [7], Fritz [5], and Schneider [17]. Our main contribution is the identification of a class of "simple" LWAA whose acceptance criterion is defined in terms of the sets of locations activated during a run, rather than the standard criterion in terms of automaton transitions. To explore the product of the transition system and the configuration graph of the LWAA, we employ a variant of Tarjan's algorithm to search for a strongly connected component that satisfies the automaton's acceptance condition.

We have implemented the proposed algorithm as an alternative verification method in the SPIN model checker [12], and we discuss some implementation options and report on experimental results. Our implementation is available for download at `http://www.pst.ifi.lmu.de/projekte/lwaaspin/`.

2 LTL and linear weak alternating automata

We define alternating ω-automata, especially LWAA, and present the translation from propositional linear-time temporal logic LTL to LWAA. Throughout, we assume a fixed finite set $\mathcal{V}$ of atomic propositions.

2.1 Linear weak alternating automata

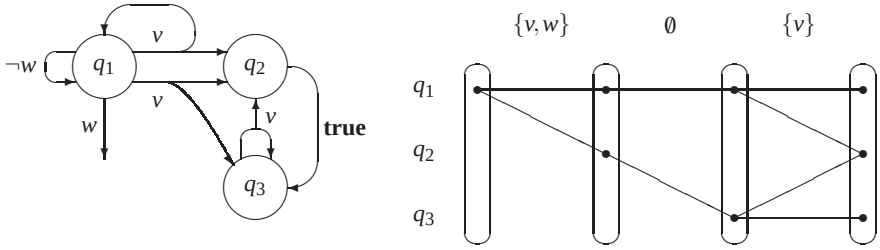
We consider automata that operate on temporal structures, i.e. ω-sequences of valuations of $\mathcal{V}$. Alternating automata combine the existential branching mode of nondeterministic automata (i.e., choice) with its dual, universal branching, where several successor locations are activated simultaneously. We present the transitions of alternating automata by associating with every location $q \in Q$ a propositional formula $\delta(q)$ over $\mathcal{V}$ and Q. For example, we interpret

$$\delta(q_1) \;=\; (v \wedge q_2 \wedge (q_1 \vee q_3)) \vee (\neg w \wedge q_1) \vee w$$

as asserting that if location q_1 is currently active and the current input satisfies v then the automaton should simultaneously activate the locations q_2 and either q_1 or q_3. If the input satisfies $\neg w$ then q_1 should be activated. If the input satisfies w then no successor locations need to be activated from q_1. Otherwise (i.e., if the input satisfies $\neg v$), the automaton blocks because the transition formula can not be satisfied. At any point during a run, a set of automaton locations (a *configuration*) will be active, and transitions are required to satisfy the transition formulas of all active locations. Locations $q \in Q$ may only occur positively in transition formulas: locations cannot be inhibited. We use the following generic definition of alternating ω-automata:

Definition 1. *An* alternating ω-automaton *is a tuple* $\mathcal{A} = (Q, q_0, \delta, Acc)$ *where*

- Q *is a finite set (of locations) where* $Q \cap \mathcal{V} = \emptyset$,
- $q_0 \in Q$ *is the initial location,*
- $\delta : Q \to \mathcal{B}(Q \cup \mathcal{V})$ *is the transition function that associates a propositional formula* $\delta(q)$ *with every location* $q \in Q$*; locations in* Q *can only occur positively in* $\delta(q)$,
- *and* $Acc \subseteq Q^\omega$ *is the acceptance condition.*



(a) Transition graph. (b) Prefix of run dag with configurations.

Fig. 1. Visualization of alternating automata and run dags.

When the transition formulas $\delta(q)$ are written in disjunctive normal form, the alternating automaton can be visualized as a hypergraph. For example, Fig. 1(a) shows an alternating ω-automaton and illustrates the above transition formula. We write $q \to q'$ if q may activate q', i.e. if q' appears in $\delta(q)$.

Runs of an alternating ω-automaton over a temporal structure $\sigma = s_0 s_1 \ldots$ are not just sequences of locations but give rise to trees, due to universal branching. However, different copies of the same target location can be identified, and we obtain a more economical dag representation as illustrated in Fig. 1(b): the vertical "slices" of the dag represent configurations that are active before reading the next input state.

We identify a set and the Boolean valuation that makes true precisely the elements of the set. For example, we say that the sets $\{v, w, q_2, q_3\}$ and $\{w\}$ satisfy the formula $\delta(q_1)$ above. For a relation $r \subseteq S \times T$, we denote its domain by $\mathrm{dom}(r)$. We denote the image of a set $A \subseteq S$ under r by $r(A)$; for $x \in S$ we sometimes write $r(x)$ for $r(\{x\})$.

Definition 2. *Let $\mathcal{A} = (Q, q_0, \delta, Acc)$ be an alternating ω-automaton and $\sigma = s_0 s_1 \ldots$, where $s_i \subseteq \mathcal{V}$, be a temporal structure. A* run dag *of $\mathcal{A}$ over σ is represented by the ω-sequence $\Delta = e_0 e_1 \ldots$ of its edges $e_i \subseteq Q \times Q$. The configurations $c_0 c_1 \ldots$ of Δ, where $c_i \subseteq Q$, are inductively defined by $c_0 = \{q_0\}$ and $c_{i+1} = e_i(c_i)$. We require that for all $i \in \mathbb{N}$, $\mathrm{dom}(e_i) \subseteq c_i$ and that for all $q \in c_i$, the valuation $s_i \cup e_i(q)$ satisfies $\delta(q)$. A* finite run dag *is a finite prefix of a run dag.*

A path *in a run dag Δ is a (finite or infinite) sequence $\pi = p_0 p_1 \ldots$ of locations $p_i \in Q$ such that $p_0 = q_0$ and $(p_i, p_{i+1}) \in e_i$ for all i. A run dag Δ is* accepting *iff $\pi \in Acc$ holds for all infinite paths π in Δ. The* language *$\mathcal{L}(\mathcal{A})$ is the set of words that admit some accepting run dag.*

Because locations do not occur negatively in transition formulas $\delta(q)$, it is easy to see that whenever $s_i \cup X$ satisfies $\delta(q)$ for some set X of locations, then so does $s_i \cup Y$ for any superset Y of X. However, the dag resulting from replacing X by Y will have more paths, making the acceptance condition harder to satisfy. It is therefore enough to consider only run dags that arise from minimal models of the transition formulas w.r.t. the states of the temporal structure, activating as few successor locations as possible.

LWAA are alternating ω-automata whose accessibility relation determines a partial order: q' is reachable from q only if q' is smaller or at most equal to q. We are interested in LWAA with a co-Büchi acceptance condition:

Definition 3. *A (co-Büchi) linear weak alternating automaton $\mathcal{A} = (Q, q_0, \delta, F)$ is a tuple where Q, q_0, and δ are as in Def. 1 and $F \subseteq Q$ is a set of locations, such that*

- *the relation $\preceq_{\mathcal{A}}$ defined by $q' \preceq_{\mathcal{A}} q$ iff $q \to^* q'$ is a partial order on Q and*
- *the acceptance condition is given by*

$$Acc = \{p_0 p_1 \ldots \in Q^\omega : p_i \in F \text{ for only finitely many } i \in \mathbb{N}\}.$$

In particular, the hypergraph of the transitions of an LWAA does not contain cycles other than self-loops, and run dags of LWAA do not contain "rising edges" as in Fig. 1. It follows that every infinite path eventually remains stable at some location q, and the acceptance condition requires that $q \notin F$ holds for that "limit location". LWAA characterize precisely the class of star-free ω-regular languages, which correspond to first-order definable ω-languages and therefore also to the languages definable by propositional LTL formulas [16,22].

2.2 From LTL to LWAA

Formulas of LTL (over atomic propositions in $\mathcal{V}$) are built using the connectives of propositional logic and the temporal operators **X** (next) and **U** (until). They are interpreted over a temporal structure $\sigma = s_0 s_1 \ldots \in (2^{\mathcal{V}})^\omega$ as follows; we write $\sigma|_i$ to denote the suffix $s_i s_{i+1} \ldots$ of σ from state s_i:

$$\begin{array}{llll}
\sigma \models p & \text{iff} \quad p \in s_0 & \sigma \models \varphi \wedge \psi & \text{iff} \quad \sigma \models \varphi \text{ and } \sigma \models \psi \\
\sigma \models \neg\varphi & \text{iff} \quad \sigma \not\models \varphi & \sigma \models \mathbf{X}\varphi & \text{iff} \quad \sigma|_1 \models \varphi \\
\sigma \models \varphi \,\mathbf{U}\, \psi & \text{iff} \quad \text{for some } i \in \mathbb{N},\ \sigma|_i \models \psi \text{ and for all } j < i,\ \sigma|_j \models \varphi & &
\end{array}$$

We freely use the standard derived operators of propositional logic and the following derived temporal connectives:

$$\begin{array}{rcll}
\mathbf{F}\varphi & \equiv & \mathbf{true}\,\mathbf{U}\,\varphi & \text{(eventually } \varphi\text{)} \\
\mathbf{G}\varphi & \equiv & \neg\mathbf{F}\neg\varphi & \text{(always } \varphi\text{)} \\
\varphi\,\mathbf{V}\,\psi & \equiv & \neg(\neg\varphi\,\mathbf{U}\,\neg\psi) & \text{(}\varphi \text{ releases } \psi\text{)}
\end{array}$$

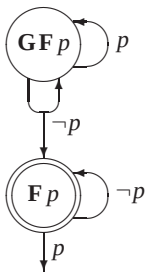
An LTL formula φ can be understood as defining the language

$$\mathcal{L}(\varphi) \;=\; \{\sigma \in (2^{\mathcal{V}})^\omega : \sigma \models \varphi\},$$

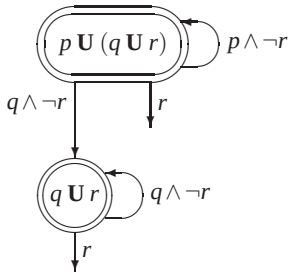
and the automata-theoretic approach to model checking builds on this identification of formulas and languages, via an effective construction of automata $\mathcal{A}_\varphi$ accepting the language $\mathcal{L}(\varphi)$. The definition of an LWAA $\mathcal{A}_\varphi$ is particularly simple [15]: without loss of generality, we assume that LTL formulas are given in negation normal form (i.e., negation is applied only to propositions), and therefore include clauses for the dual operators $\vee$ and **V**. The automaton is $\mathcal{A}_\varphi = (Q, q_\varphi, \delta, F)$ where Q contains a location q_ψ

location q	$\delta(q)$
q_ψ (ψ a literal)	ψ
$q_{\psi\wedge\chi}$	$\delta(q_\psi)\wedge\delta(q_\chi)$
$q_{\psi\vee\chi}$	$\delta(q_\psi)\vee\delta(q_\chi)$
$q_{\mathbf{X}\psi}$	q_ψ
$q_{\psi\mathbf{U}\chi}$	$\delta(q_\chi)\vee(\delta(q_\psi)\wedge q_{\psi\mathbf{U}\chi})$
$q_{\psi\mathbf{V}\chi}$	$\delta(q_\chi)\wedge(\delta(q_\psi)\vee q_{\psi\mathbf{V}\chi})$

(a) Transition formulas of $\mathcal{A}_\varphi$



(b) $\mathcal{A}_{\mathbf{GF}\,p}$



(c) $\mathcal{A}_{p\mathbf{U}(q\mathbf{U}r)}$

Fig. 2. Translation of LTL formulas into LWAA.

for every subformula ψ of φ, with q_φ being the initial location. The transition formulas $\delta(q_\psi)$ are defined in Fig. 2(a); in particular, LTL operators are simply decomposed according to their fixpoint characterizations. The set F of co-final locations consists of all locations $q_{\psi\mathbf{U}\chi}\in Q$ that correspond to "until" subformulas of φ. It is easy to verify that the resulting automaton $\mathcal{A}_\varphi$ is an LWAA: for any locations q_ψ and q_χ, the definition of $\delta(q_\psi)$ ensures that $q_\psi\rightarrow q_\chi$ holds only if χ is a subformula of ψ. Correctness proofs for the construction can be found in [15,23]; conversely, Rohde [16] and Löding and Thomas [14] prove that for every LWAA $\mathcal{A}$ there is an LTL formula $\varphi_\mathcal{A}$ such that $\mathcal{L}(\varphi_\mathcal{A})=\mathcal{L}(\mathcal{A})$.

The number of subformulas of an LTL formula φ is linear in the length of φ, and therefore so is the size of $\mathcal{A}_\varphi$. However, in practice the automaton should be minimized further. Clearly, unreachable locations can be eliminated. Moreover, whenever there is a choice between activating sets X or Y of locations where $X\subseteq Y$ from some location q, the smaller set X should be preferred, and Y should be activated only if X cannot be. As a simple example, we can define $\delta(q_{\mathbf{F}\,p})=p\vee(\neg p\wedge q_{\mathbf{F}\,p})$ instead of $\delta(q_{\mathbf{F}\,p})=p\vee q_{\mathbf{F}\,p}$.

Figure 2 shows two linear weak alternating automata obtained from LTL formulas by applying this construction (the locations in F are indicated by double circles).

Further minimizations are less straightforward. Because the automaton structure closely resembles the structure of the LTL formula, heuristics to minimize the LTL formula [4,18] are important. Fritz and Wilke [6] discuss more elaborate optimizations based on simulation relations on the set Q of locations.

3 Deciding language emptiness for LWAA

In general, it is nontrivial to decide language emptiness for alternating ω-automata, due to their intricate combinatorial structure: a configuration consists of a set of automaton locations that have to "synchronize" on the current input state during a transition to a successor configuration. The standard approach is therefore based on a translation to non-deterministic Büchi automata, for which emptiness can be decided in linear time. Unfortunately, this translation is of exponential complexity.

Linear weak alternating automata have a simpler combinatorial structure: the transition graph contains only trivial cycles, and therefore a run dag is non-accepting only if it contains a path that ends in a self-loop at some location $q \in F$. This observation gives rise to the following non-emptiness criterion for LWAA, which is closely related to Theorem 2 of [7]:

Theorem 4. *Assume that $\mathcal{A} = (Q, q_0, \delta, F)$ is an LWAA. Then $\mathcal{L}(\mathcal{A}) \neq \emptyset$ if and only if there exists a finite run dag $\Delta = e_0 e_1 \dots e_n$ with configurations $c_0 c_1 \dots c_{n+1}$ over a finite sequence $s_0 \dots s_n$ of states and some $k \leq n$ such that*

1. *$c_k = c_{n+1}$ and*
2. *for every $q \in F$, one has $(q,q) \notin e_j$ for some j where $k \leq j \leq n$.*

Proof. "If": Consider the infinite dag $\Delta' = e_0 \dots e_{k-1}(e_k \dots e_n)^\omega$. Because $c_k = c_{n+1}$, it is obvious that Δ' is a run dag over $\sigma = s_0 \dots s_{k-1}(s_k \dots s_n)^\omega$; we now show that Δ' is accepting. Assume, to the contrary, that $\pi = p_0 p_1 \dots$ is some infinite path in Δ' such that $p_i \in F$ holds for infinitely many $i \in \mathbb{N}$. Because $\mathcal{A}$ is an LWAA, there exists some $m \in \mathbb{N}$ and some $q \in Q$ such that $p_i = q$ for all $i \geq m$. It follows that $(q,q) \in e_i$ holds for all $i \geq m$, which is impossible by assumption (2) and the construction of Δ'. Therefore, Δ' must be accepting, and $\mathcal{L}(\mathcal{A}) \neq \emptyset$.

"Only if": Assume that $\sigma = s_0 s_1 \dots \in \mathcal{L}(\mathcal{A})$, and let $\Delta' = e_0 e_1 \dots$ be some accepting run dag of $\mathcal{A}$ over σ. Since Q is finite, Δ' can contain only finitely many different configurations $c_0, c_1, \dots$, and there is some configuration $c \subseteq Q$ such that $c_i = c$ for infinitely many $i \in \mathbb{N}$. Denote by $i_0 < i_1 < \dots$ the ω-sequence of indexes such that $c_{i_j} = c$. If there were some $q \in F$ such that $q \in e_j(q)$ for all $j \geq i_0$ (implying in particular that $q \in c_j$ for all $j \geq i_0$ by Def. 2) then Δ' would contain an infinite path ending in a self-loop at q, contradicting the assumption that Δ' is accepting. Therefore, for every $q \in F$ there must be some $j_q \geq i_0$ such that $(q,q) \notin e_{j_q}$. Choosing $k = i_0$ and $n = i_m - 1$ for some m such that $i_m > j_q$ for all (finitely many) $q \in F$, we obtain a finite run dag Δ as required. □

Observe that Thm. 4 requires to inspect the *transitions* of the dag and not just the configurations. In fact, a run dag may well be accepting although some location $q \in F$ is contained in all (or almost all) configurations. For example, consider the LWAA for the formula $\mathbf{GXF}\,p$: the location $q_{\mathbf{F}\,p}$ will be active in every run dag from the second configuration onward, even if the run dag is accepting. We now introduce a class of LWAA for which it is enough to inspect the configurations.

Definition 5. *An LWAA $\mathcal{A} = (Q, q_0, \delta, F)$ is* simple *if for all $q \in F$, all $q' \in Q$, all states $s \subseteq \mathcal{V}$, and all $X, Y \subseteq Q$ not containing q, if $s \cup X \cup \{q\} \models \delta(q')$ and $s \cup Y \models \delta(q)$ then $s \cup X \cup Y \models \delta(q')$.*

In other words, if a co-final location q can be activated from some location q' for some state s while it can be exited during the same transition, then q' has an alternative transition that avoids activating q, and this alternative transitions activates only locations that would anyway have been activated by the joint transitions from q and q'. For simple LWAA, non-emptiness can be decided on the basis of the visited configurations alone, without memorizing the graph structure of the run dag.

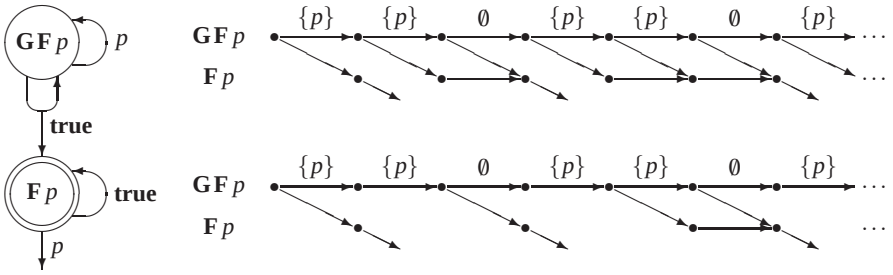


Fig. 3. Illustration of the construction of Thm. 6.

Theorem 6. *Assume that $\mathcal{A} = (Q, q_0, \delta, F)$ is a simple LWAA. Then $\mathcal{L}(\mathcal{A}) \neq \emptyset$ if and only if there exists a finite run dag $\Delta = e_0 e_1 \dots e_n$ with configurations $c_0 c_1 \dots c_{n+1}$ over a finite sequence $s_0 \dots s_n$ of states and some $k \leq n$ such that*

1. *$c_k = c_{n+1}$ and*
2. *for every $q \in F$, one has $q \notin c_j$ for some j where $k \leq j \leq n$.*

Proof. "If": The assumption $q \notin c_j$ and the requirement that $\mathrm{dom}(e_j) \subseteq c_j$ imply that $(q,q) \notin e_j$, and therefore $\mathcal{L}(\mathcal{A}) \neq \emptyset$ follows using Thm. 4.

"Only if": Assume that $\mathcal{L}(\mathcal{A}) \neq \emptyset$, obtain a finite run dag Δ satisfying the conditions of Thm. 4, and let $l = n - k + 1$ denote the length of the loop. "Unwinding" Δ, we obtain an infinite run dag $e_0 e_1 \dots$ over the temporal structure $s_0 s_1 \dots$ whose edges are $e_i = e_{k+((i-k) \bmod l)}$ for $i > n$, and similarly for the states s_i and the configurations c_i. W.l.o.g. we assume that the dag contains no unnecessary edges, i.e. that for all $e_i \in \Delta$, $(q,q') \in e_i$ holds only if $q \to q'$.

We inductively construct an infinite run dag $\Delta' = e'_0 e'_1 \dots$ with configurations $c'_0 c'_1 \dots$ such that $c'_i \subseteq c_i$ as follows: let $c'_0 = c_0$ and for $i < k$, let $e'_i = e_i$ and $c'_{i+1} = c_{i+1}$. For $i \geq k$, assume that c'_i has already been defined. Let F_i denote the set of $q \in c'_i \cap F$ such that $(q,q) \notin e_i$ but $q \in e_i(c'_i)$, and for any $q \in F_i$ let Q'_q denote the set of locations $q' \in c'_i$ such that $(q',q) \in e_i$ and let $Y_q = e_i(q)$. Because $\mathcal{A}$ is simple, it follows that $s_i \cup (e_i(q') \setminus \{q\}) \cup Y_q \models \delta(q')$, for all $q \in F_i$ and $q' \in Q'_q$. We let e'_i be obtained from the restriction of e_i to c'_i by deleting all edges (q',q) for $q \in F_i$ and adding edges (q',q'') for all $q' \in Q'_q$ and $q'' \in Y_q$, for $q \in F_i$. Clearly, this ensures that $c'_{i+1} \subseteq c_{i+1}$ holds for the resulting configuration and that $c'_{i+1} \cap F_i = \emptyset$.

For any $q \in F_i$, the definition of an LWAA and the assumption that $q \notin Y_q$ ensure that $q'' \prec_{\mathcal{A}} q$ holds for all $q'' \in Y_q$, as well as $q \preceq_{\mathcal{A}} q'$ for all $q' \in Q'_q$. In particular, we must have $q'' \neq q'$ for all $q'' \in Y_q$ and $q' \in Q'_q$, and therefore e'_i does not contain more self loops than e_i: for all $p \in Q$, we have $(p,p) \in e'_i$ only if $(p,p) \in e_i$.

Consequently, Δ' is an accepting infinite run dag such that for every $q \in F$ there exists some $j \geq k$ such that $q \notin c'_j$. It now suffices to pick some $n \geq k$ satisfying the conditions of the theorem; such an n exists because F is finite and Δ' can contain only finitely many different configurations. □

Fig. 3 illustrates two accepting run dags for a simple LWAA: the dag shown above satisfies the criterion of Thm. 4 although the co-final location corresponding to $\mathbf{F}\,p$

remains active from the second configuration onward. The dag shown below is the result of the transformation described in the proof, and indeed the location $\mathbf{F}\,p$ is infinitely often inactive.

We now show that the LWAA $\mathcal{A}_\varphi$ for an LTL formula φ is simple provided φ does not contain subformulas $\mathbf{X}(\chi\ \mathbf{U}\ \chi')$. Such subformulas are easily avoided because $\mathbf{X}$ distributes over $\mathbf{U}$. Actually, our implementation exploits the commutativity of $\mathbf{X}$ with all LTL connectives to rewrite formulas such that no other temporal operators are in the scope of $\mathbf{X}$; this is useful for preliminary simplifications at the formula level. Also, the transformations described at the end of Sect. 2.2 ensure that the LWAA remains simple.

Theorem 7. *For any LTL formula φ that does not contain any subformula $\mathbf{X}(\chi\ \mathbf{U}\ \chi')$, the automaton $\mathcal{A}_\varphi$ is a simple LWAA.*

Proof. Let $\mathcal{A}_\varphi = (Q, q_\varphi, \delta, F)$ and assume that $q \in F$, $q' \in Q$, and $X, Y \subseteq Q$ are as in Def. 5, in particular $s \cup X \cup \{q\} \models \delta(q')$ and $s \cup Y \models \delta(q)$. The proof is by induction on ψ where $q' = q_\psi$.

- $\psi \equiv (\neg)v:\ \delta(q') = \psi$, so we must have $s \models \delta(q')$, and the assertion $s \cup X \cup Y \models \delta(q')$ follows trivially.
- $\psi \equiv \chi \otimes \chi',\ \otimes \in \{\wedge, \vee\}:\ \delta(q') = \delta(q_\chi) \otimes \delta(q_{\chi'})$, and the assertion follows easily from the induction hypothesis.
- $\psi \equiv \mathbf{X}\chi:\ \delta(q') = q_\chi$, and by assumption χ is not an $\mathbf{U}$ formula, so $q_\chi \notin F$. In particular, $q_\chi \neq q$, and so the assumption $s \cup X \cup \{q\} \models \delta(q')$ implies that $s \cup X \models \delta(q')$, and the assertion $s \cup X \cup Y \models \delta(q')$ follows by monotonicity.
- $\psi \equiv \chi\ \mathbf{U}\ \chi':\ \delta(q') = \delta(q_{\chi'}) \vee (\delta(q_\chi) \wedge q')$. In case $s \cup X \cup \{q\} \models \delta(q_{\chi'})$, the induction hypothesis implies $s \cup X \cup Y \models \delta(q_{\chi'})$, hence also $s \cup X \cup Y \models \delta(q')$.
  If $s \cup X \cup \{q\} \models \delta(q_\chi) \wedge q'$, we consider two cases: if $q = q'$ then $s \cup Y \models \delta(q')$ holds by assumption. Moreover, $s \cup X \cup Y \models \delta(q_\chi)$ holds by induction hypothesis, and the assertion follows.
  Otherwise, we must have $q' \in X$. Again, $s \cup X \cup Y \models \delta(q_\chi)$ follows from the induction hypothesis, and since $q' \in X$ it follows that $s \cup X \cup Y \models \delta(q_\chi) \wedge q'$.
- $\psi \equiv \chi\ \mathbf{V}\ \chi':\ \delta(q') = \delta(q_{\chi'}) \wedge (\delta(q_\chi) \vee q')$. In particular, $s \cup X \cup \{q\} \models \delta(q_{\chi'})$, and we obtain $s \cup X \cup Y \models \delta(q_{\chi'})$ by induction hypothesis.
  If $s \cup X \cup \{q\} \models \delta(q_\chi)$, we similarly obtain $s \cup X \cup Y \models \delta(q_\chi)$. Otherwise, note that $q \neq q'$ because $q \in F$ and $q' \notin F$ (since it is not an $\mathbf{U}$ formula). Therefore, we must have $s \cup X \models q'$, and a fortiori $s \cup X \cup Y \models q'$, completing the proof. □

Let us note in passing that simple LWAA are as expressive as LWAA, i.e. they also characterize the class of star-free ω-regular languages: from [14,16] we know that for every LWAA $\mathcal{A}$ there is an LTL formula $\varphi_\mathcal{A}$ such that $\mathcal{L}(\varphi_\mathcal{A}) = \mathcal{L}(\mathcal{A})$. Since $\mathbf{X}$ distributes over $\mathbf{U}$, $\varphi_\mathcal{A}$ can be transformed into an equivalent formula φ' of the form required in Thm. 7, and $\mathcal{A}_{\varphi'}$ is a simple LWAA accepting the same language as $\mathcal{A}$.

4 Model checking algorithm

We describe a model checking algorithm based on the nonemptiness criterion of Thm. 6, and we discuss some design decisions encountered in our implementation. The algorithm has been integrated within the LTL model checker SPIN, and we present some results that have been obtained on benchmark examples.

```
procedure Visit(s, C):
  let c = (s,C) in
    inComp[c] := false; root[c] := c; labels[c] := ∅;
    cnt[c] := cnt; cnt := cnt+1; seen := seen ∪ {c};
    push(c, stack);
    forall c′ = (s′,C′) in Succ(c) do
      if c′ ∉ seen then Visit(s′,C′) end if;
      if ¬inComp[c′] then
        if cnt[root[c′]] < cnt[root[c]] then
          labels[root[c′]] := labels[root[c′]] ∪ labels[root[c]];
          root[c] := root[c′]
        end if;
        labels[root[c]] := labels[root[c]]
                           ∪ (f_lwaa \ C); // f_lwaa ≡ co-final locations
        if labels[root[c]] = f_lwaa then raise Good_Cycle end if;
      end if;
    end forall;
    if root[c]=c then
      repeat
        d := pop(stack);
        inComp[d] := true;
      until d=c;
    end if;
  end let;
end Visit;

procedure Check:
  stack := empty; seen := ∅; cnt := 0;
  Visit(init_ts, {init_lwaa});    // start with initial location
end Check;
```

Fig. 4. LWAA-based model checking algorithm.

4.1 Adapting Tarjan's algorithm

Theorem 6 contains the core of our model checking algorithm: given the simple LWAA $\mathcal{A}_{\neg\varphi}$ corresponding to the negation $\neg\varphi$ of the property to be verified, we explore the product of the transition system $\mathcal{T}$ and the graph of configurations of $\mathcal{A}_{\neg\varphi}$, searching for a strongly connected component that satisfies the acceptance condition. In fact, in the light of Thm. 6 a simple LWAA $\mathcal{A}$ can alternatively be viewed as a symbolic representation of a GBA whose locations are sets of locations of $\mathcal{A}$, and that has an acceptance condition per co-final location of $\mathcal{A}$.

The traditional CVWY algorithm [2] for LTL model checking based on Büchi automata has been generalized for GBA by Tauriainen [21], but we find it easier to adapt Tarjan's algorithm [19] for finding strongly connected components in graphs. Figure 4 gives a pseudo-code representation of our algorithm. The depth-first search operates on pairs (s,C) where s is a state of the transition system and C is a configuration of the LWAA. Given a pair $c = (s,C)$, the call to `Succ` computes the set $succ_{\mathcal{T}}(s) \times succ_{\mathcal{A}}(s,C)$ containing all pairs $c' = (s',C')$ of successor states s' of the transition system and successor configurations C' of the LWAA, i.e. those C' which satisfy $s \cup C' \models \delta(q)$ for all $q \in C$. Tarjan's algorithm assigns a so-called root candidate `root` to each node of the graph, which is the oldest node on the stack known to belong to the same SCC.

In model checking, we are not so much interested in actually computing SCCs: it is sufficient to verify that the acceptance criterion of Thm. 6 is met for some strongly connected subgraph (SCS). To do so, we associate a `labels` field with the root candidate of each SCC that accumulates the locations $q \in F$ that have been found absent in some pair (s,C) contained in the SCC. Whenever `labels` is found to contain all co-final states of the LWAA (denoted by `f_lwaa`), the SCS must be accepting and the search is aborted. Note that we need to maintain two stacks: one for the depth-first search recursion, and one for identifying SCCs.

If an accepting SCS is found, we also want to produce a counter-example, and Tarjan's algorithm is less convenient for this purpose than the CVWY algorithm whose recursion stack contains the counter-example once a cycle has been detected. In our case, neither the recursion stack nor the SCC stack represent a complete counter-example. A counter-example can still be obtained by traversing the nodes of an accepting SCS that have already been visited, without re-considering the transition system. We add two pointers to our node representation in the SCC stack, representing "backward" and "forward" links that point to the pair from which the current node was reached and to the oldest pair on the stack that is a successor of the current pair. Indeed, one can show that the subgraph of nodes on the SCC stack with neighborhood relation

$$\{(c,c') : c' = \mathit{forward}(c) \text{ or } c = \mathit{backward}(c')\}$$

also forms an SCS of the product graph. A counter-example can now be produced by enforcing a visit to all the pairs that satisfy some acceptance condition.

4.2 Computation of successor configurations

The efficient generation of successor configurations in $\mathit{succ}_{\mathcal{A}}(s,C)$ is a crucial part of our algorithm. Given a configuration $C \subseteq Q$ of the LWAA and a state s of the transition system (which we identify with a valuation of the propositional variables), we need to compute the set of all C' such that $s \cup C' \models \delta(q)$ holds for all $q \in C$. Moreover, we are mainly interested in finding minimal successor configurations.

An elegant approach towards computing successor configurations makes use of BDDs [1]. In fact, the transitions of an LWAA can be represented by a single BDD. The set of minimal successor configurations is obtained by conjoining this BDD with the BDD representations of the state s and the source configuration C, and then extracting the set of all satisfying valuations of the resulting BDD. Some experimentation convinced us, however, that the resulting BDDs become too big for large LTL formulas. Alternatively, one can store BDDs representing $\delta(q)$ for each location q and form the conjunction of all $\delta(q)$ for $q \in C$. Again, this approach turned out to consume too much memory.

We finally resorted to using BDDs only as a representation of configurations. To do so, we examine the hyperedges of the transition graph of the LWAA, which correspond to the clauses of the disjunctive normal form of $\delta(q)$. For every location $q \in C$, we compute the disjunction of its enabled transitions, and then take the conjunction over all locations in C. We thus obtain

$$\mathit{succ}_{\mathcal{A}}(s,C) \;=\; \bigwedge_{q \in C} \Big(\bigvee_{t \in \mathit{enabled}(s,q)} (t \setminus \mathcal{V}) \Big)$$

as the BDD representing the set of successor configurations, where $enabled(s,q)$ denotes the set of enabled transitions of q for state s, i.e. those transitions t for which $s \cup Q \models t$. Although this requires pre-computing a potentially exponentially large set of transitions, this approach appears to be fastest for BDD-based calculation of successor nodes.

We compare this approach to a direct calculation of successor configurations that stores them as a sorted list, which is pruned to remove non-minimal successors. Although the pruning step is of quadratic complexity in our implementation (it could be improved to $O(n \log n)$ time), experiments showed that it pays off handsomely because fewer nodes need to be explored in the graph search.

4.3 Adapting Spin

Either approach to computing successors works best if we can efficiently determine the set of enabled transitions of an LWAA location. One way to do this is to generate C source code for a given LWAA and then use the CPU arithmetics. The SPIN model checker employs a similar approach, albeit for Büchi automata, and this is one of reasons why we adapted it to use our algorithm.

SPIN [10,12], is generally considered as one of the fastest and most complete tools for protocol verification. For a given model (written in Promela) and Büchi automaton (called "never-claim"), it generates C sources that are then compiled to produce a model-specific model checker. SPIN also includes a translation from LTL formulas to Büchi automata, but for our comparisons we used the LTL2BA tool due to Gastin and Oddoux [7], which is faster by orders of magnitude for large LTL formulas.

Our adaptation, called LWAASPIN, adds the generation of LWAA to SPIN, and modifies the code generation to use Tarjan's algorithm and on-the-fly calculation of successor configurations. This involved about 150 code changes, and added about 2600 lines of code. SPIN includes elaborate optimizations, such as partial-order reduction, that are independent of the use of non-deterministic or alternating automata and that can therefore be used with our implementation as well. We have not yet adapted SPIN's optimizations of memory usage such as bitstate hashing to our algorithm, although we see no obstacle in principle to do so.

4.4 Experimental results

Geldenhuys and Valmari [8] have recently proposed to use Tarjan's algorithm, but for non-deterministic Büchi automata, and we have implemented their algorithm for comparison. We have not been able to reproduce their results indicating that Tarjan's algorithm outperforms the CVWY algorithm on nondeterministic Büchi automata (their paper does not indicate which implementation of CVWY was used). In our experiments, both algorithms perform head-to-head on most examples. We now describe the results for the implementation based on LWAA.

For most examples, the search for an accepting SCS in the product graph is slower than the runtime of the model checker produced by SPIN after LTL2BA has generated the Büchi automaton. However, our algorithm can be considerably faster than generating the Büchi automaton and then checking the emptiness of the product automaton,

for large LTL formulas. However, note that both SPIN and our implementation use unguided search, and we can thus not exactly compare single instances of satisfiable problems.

Large LTL formulas are not as common as one might expect. SPIN's implementation of the CVWY algorithm can handle weak fairness of processes directly; such conditions do not have to be added to the LTL formula to be verified. We present two simple and scalable examples: the dining philosophers problem and a binary semaphore protocol.

For the dining philosophers example, we want to verify that if every philosopher holds exactly one fork infinitely often, then philosopher 1 will eventually eat:

$$\mathbf{GF}\,\mathit{hasFork}_1 \wedge \ldots \wedge \mathbf{GF}\,\mathit{hasFork}_n \;\Rightarrow\; \mathbf{GF}\,\mathit{eat}_1$$

The model `dinphil`*n* denotes the situation where all *n* philosophers start with their right-hand fork, which may lead to a deadlock. The model `dinphil`*n*`i` avoids the deadlock by letting the *n*-th philosopher start with his left-hand fork.

For the binary semaphore example we claim that if strong fairness is ensured for each process, all processes will eventually have been in their critical section:

$$(\mathbf{GF}\,\mathit{canenter}_1 \Rightarrow \mathbf{GF}\,\mathit{enter}_1) \wedge \ldots \wedge (\mathbf{GF}\,\mathit{canenter}_n \Rightarrow \mathbf{GF}\,\mathit{enter}_n) \;\Rightarrow\; \mathbf{F}\,\mathit{allcrit}$$

By `sfgood`*n*, we denote a constellation with *n* processes and strong fairness assumed for each of them, while `sfbad`*n* denotes the same constellation, except with weak fairness for process p_n, which will allow the process to starve.

Table 1 contains timings (in seconds) for the different steps of the verification process for SPIN 4.1.1 and for our LWAASPIN implementation. SPIN requires successive invocations of `ltl2ba`, `spin`, `gcc` and `pan`; LWAASPIN combines the first two stages. The times were measured on an Intel Pentium® 4, 3.0 GHz computer with 1GB main memory running Linux and without other significant process activity. Entries "o.o.t." indicate that the computation did not finish within 2 hours, while "o.o.m." means "out of memory".

We can see that most of the time required by SPIN is spent on preparing the `pan` model checker, either by calculating the non-deterministic Büchi automata for the dining philosophers, or by handling the large automata sources for the binary semaphore example. LWAASPIN significantly reduces the time taken for pre-processing.

The sizes of the generated automata are indicated in Tab. 2. "States seen" denotes the number of distinct states (of the product automaton) encountered by LWAASPIN using the direct successor configuration calculation approach. It should be noted that the Büchi automata for the dining philosophers example are very small compared to the size of the formula, and are in fact linear; even for the `dinphil10i` case, the automaton contains only 12 locations. This is not true for the semaphore example: the Büchi automaton for `sfgood7` contains 3025 locations and 23391 transitions. Still, one advantage of using LTL2BA is that a Büchi automaton that has been computed once can be stored and reused; this could reduce the overall verification time for the dining philosophers example where the same formula is used for both the valid and the invalid model.

We can draw two conclusions from our data: first, the preprocessing by `lwaaspin` uses very little time because we do not have to calculate the Büchi automaton (although

Problem	Counter-example	SPIN				LWAASPIN		
		ltl2ba	spin	gcc	pan	lwaaspin	gcc	pan
dinphil6	yes	0.431	0.019	0.601	0.079	0.019	0.579	0.163
dinphil8	yes	35.946	0.02	0.671	0.133	0.027	0.818	0.166
dinphil10	yes	3611.724	0.025	0.767	1.642	0.057	1.899	0.170
dinphil12	yes	o.o.t.				0.141	6.644	0.206
dinphil14	yes					0.499	28.082	0.431
dinphil15	yes					0.972	o.o.m.	
dinphil6i	no	0.431	0.024	0.639	0.244	0.020	0.616	0.569
dinphil8i	no	35.946	0.021	0.711	7.309	0.028	0.861	20.177
dinphil10i	no	3611.724	0.025	0.807	722.874	0.070	2.623	623.760
dinphil11i	no	o.o.t.				0.099	3.438	o.o.m.
sfbad6	yes	1.904	0.912	7.284	0.025	0.066	2.211	1.312
sfbad7	yes	27.674	42.525	o.o.m.		0.179	7.423	7.848
sfbad8	yes					0.784	43.472	7.000
sfbad9	yes					2.627	o.o.m.	
sfgood6	no	2.292	17.329	27.608	2.193	0.064	2.227	2.540
sfgood7	no	36.306	417.485	o.o.m.		0.357	8.214	15.940
sfgood8	no					0.718	42.688	140.130
sfgood9	no					2.634	o.o.m.	

Table 1. Comparison of SPIN and LWAASPIN (BDD-less successor calculation)

Problem	Successor calculation		LWAA		Büchi		States seen
	BDD	direct	Locations	Transitions	Locations	Transitions	
dinphil6	0.834	0.761	10	207	8	36	105
dinphil8	1.194	1.011	12	787	10	55	119
dinphil10	2.803	2.126	14	3095	12	78	133
dinphil6i	1.291	1.205	10	207	8	36	46165
dinphil8i	21.802	21.021	12	787	10	55	$1.2 \cdot 10^6$
dinphil10i	643.006	626.453	14	3095	12	78	$1.5 \cdot 10^7$
sfbad6	16.664	3.589	26	4140	252	1757	137882
sfbad7	354.874	15.461	30	16435	1292	8252	597686
sfgood6	32.261	4.831	26	4139	972	5872	221497
sfgood7	115.539	24.511	30	16434	3025	23391	872589

Table 2. Comparison of successor calculation, and sizes of the automata.

strictly speaking our implementation is also exponential because it transforms the transition formulas into disjunctive normal form). This makes up for the usually inferior performance of our pan version. It also means that we can at least start a model checking run, even for very large LTL formulas, in the hope of finding a counter-example. Second, we can check larger LTL formulas. Ultimately, we encounter the same difficulties as SPIN during both the gcc and the pan phases; after all, we are confronted with a PSPACE-complete problem. The pre-processing phase could be further reduced by avoiding the generation of an exponential number of transitions in the C sources,

postponing more work to the pan executable. Besides, the bitstate hashing technique as implemented in SPIN [11] could also be applied to Tarjan's algorithm.

Table 2 also compares the two approaches to computing successor configurations described in Sect. 4.2. The BDD-based approach appears to be less predictable and never outperforms the direct computation, but further experience is necessary to better understand the tradeoff.

5 Conclusion and further work

We have presented a novel algorithm for the classical problem of LTL model checking. It uses an LWAA encoding of the LTL property as a symbolic representation of the corresponding GBA, which is effectively generated on the fly during the state space search, and never has to be stored explicitly. By adapting the SPIN model checker to our approach, we validate that, for large LTL formulas, the time gained by avoiding the expensive construction of a non-deterministic Büchi automaton more than makes up for the runtime penalty due to the implicit GBA generation during model checking, and this advantage does not appear to be offset by the simplifications applied to the intermediate automata by algorithms such as LTL2BA. However, we do not yet really understand the relationship between minimizations at the automaton level and the local optimizations applied in our search.

We believe that our approach opens the way to verifying large LTL formulas by model checking. Further work should investigate the possibilities that arise from this opportunity, such as improving techniques for software model checking based on predicate abstraction. Also, our implementation still leaves room for performance improvements. In particular, the LWAA should be further minimized, the representation of transitions could be reconsidered, and the memory requirements could be reduced by clever coding techniques.

References

1. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, C-35(8):677–691, 1986.
2. C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal methods in system design*, 1:275–288, 1992.
3. M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In N. Halbwachs and D. Peled, editors, *11th Intl. Conf. Computer Aided Verification (CAV'99)*, volume 1633 of *Lect. Notes in Comp. Sci.*, pages 249–260, Trento, Italy, 1999. Springer-Verlag.
4. K. Etessami and G. Holzmann. Optimizing Büchi automata. In C. Palamidessi, editor, *CONCUR 2000 - Concurrency Theory: 11th International Conference*, volume 1877 of *Lect. Notes in Comp. Sci.*, pages 153–167, University Park, PA, 2000. Springer-Verlag.
5. C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In O. Ibarra and Z. Dang, editors, *8th Intl. Conf. Implementation and Application of Automata (CIAA 2003)*, volume 2759 of *Lect. Notes in Comp. Sci.*, pages 35–48, Santa Barbara, CA, USA, 2003. Springer-Verlag.

6. C. Fritz and T. Wilke. State space reductions for alternating Büchi automata: Quotienting by simulation equivalences. In M. Agrawal and A. Seth, editors, *22nd Conf. Found. Software Tech. and Theor. Comp. Sci. (FSTTCS 2002)*, volume 2556 of *Lect. Notes in Comp. Sci.*, pages 157–168, Kanpur, India, 2002. Springer-Verlag.
7. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *13th Intl. Conf. Computer Aided Verification (CAV'01)*, volume 2102 of *Lect. Notes in Comp. Sci.*, pages 53–65, Paris, France, 2001. Springer-Verlag.
8. J. Geldenhuys and A. Valmari. Tarjan's algorithm makes LTL verification more efficient. In K. Jensen and A. Podelski, editors, *10th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lect. Notes in Comp. Sci.*, pages 205–219, Barcelona, Spain, 2004. Springer-Verlag.
9. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
10. G. Holzmann. The Spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
11. G. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.
12. G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.
13. Y. Kesten and A. Pnueli. Verifying liveness by augmented abstraction. In J. Flum and M. Rodríguez-Artalejo, editors, *Computer Science Logic (CSL'99)*, volume 1683 of *Lect. Notes in Comp. Sci.*, pages 141–156, Madrid, Spain, 1999. Springer-Verlag.
14. C. Löding and W. Thomas. Alternating automata and logics over infinite words. In J. van Leeuwen et al., editor, *IFIP Intl. Conf. Theor. Comp. Sci. (TCS 2000)*, volume 1872 of *Lect. Notes in Comp. Sci.*, pages 521–535, Sendai, Japan, 2000.
15. D.E. Muller, A. Saoudi, and P.E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *3rd IEEE Symp. Logic in Computer Science (LICS'88)*, pages 422–427, Edinburgh, Scotland, 1988. IEEE Press.
16. S. Rohde. *Alternating automata and the temporal logic of ordinals*. PhD thesis, Dept. of Math., Univ. of Illinois, Urbana-Champaign, IL, 1997.
17. K. Schneider. Yet another look at LTL model checking. In L. Pierre and T. Kropf, editors, *IFIP Work. Conf. Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lect. Notes in Comp. Sci.*, pages 321–326, Bad Herrenalb, Germany, 1999. Springer-Verlag.
18. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E.A. Emerson and A.P. Sistla, editors, *12th Intl. Conf. Computer Aided Verification (CAV 2000)*, volume 1633 of *Lect. Notes in Comp. Sci.*, pages 257–263, Chicago, IL, 2000. Springer-Verlag.
19. R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.
20. H. Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Research Report A83, Helsinki Univ. of Technology, Lab. Theor. Comp. Sci., Espoo, Finland, December 2003.
21. H. Tauriainen. Nested emptiness search for generalized Büchi automata. In M. Kishnevsky and Ph. Darondeau, editors, *4th Intl. Conf. Application of Concurrency to System Design (ACSD 2004)*, pages 165–174, Hamilton, Ontario, 2004. IEEE Computer Society.
22. W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer-Verlag, 1997.
23. M. Y. Vardi. Alternating automata and program verification. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lect. Notes in Comp. Sci.*, pages 471–485. Springer-Verlag, 1995.