Larger Automata and Less Work for LTL Model Checking

Jaco Geldenhuys[1] and Henri Hansen[2]

[1] Department of Computer Science, Stellenbosch University,
Private Bag X1, 7602 Matieland, South Africa
jaco@cs.sun.ac.za
[2] Institute of Software Systems, Tampere University of Technology,
PO Box 553, FI-33101 Tampere, Finland
hansen@cs.tut.fi

Abstract. Many different automata and algorithms have been investigated in the context of automata-theoretic LTL model checking. This article compares the behaviour of two variations on the widely used Büchi automaton, namely (i) a Büchi automaton where states are labelled with atomic propositions and transitions are unlabelled, and (ii) a form of testing automaton that can only observe changes in state propositions and makes use of special livelock acceptance states. We describe how these variations can be generated from standard Büchi automata, and outline an SCC-based algorithm for verification with testing automata.

The variations are compared to standard automata in experiments with both random and human-generated Kripke structures and LTL$_{-X}$ formulas, using SCC-based algorithms as well as a recent, improved version of the classic nested search algorithm. The results show that SCC-based algorithms outperform their nested search counterpart, but that the biggest improvements come from using the variant automata.

Much work has been done on the generation of small automata, but small automata do not necessarily lead to small products when combined with the system being verified. We investigate the underlying factors for the superior performance of the new variations.

1 Introduction

The automata-theoretic approach to model checking is based on the correspondence between temporal logic, automata and formal languages. Checking that a system S complies with a temporal logic correctness formula entails the application of two algorithms: the first to translate a formula ϕ to an ω-automaton (on infinite words), and the second to determine whether the intersection of this automaton and a similar automaton derived directly from S accepts only the empty language. It comes as no surprise that since this approach was first proposed, the use of many different kinds of automata has been investigated, and several variations on the two algorithms have been proposed; some of this work is mentioned in Section 2.

A. Valmari (Ed.): SPIN 2006, LNCS 3925, pp. 53–70, 2006.

It is probably accurate to say that most of the research in this field is based on Büchi automata with propositional formulas on transitions. We shall refer to this standard form as *transition-labelled*. In this work we study two variations on this theme. First, in Section 3, we consider Büchi automata where the states carry propositional formulas and the transitions are unlabelled — we shall refer to these as *state-labelled* Büchi automata. The second form, the so-called *testing automaton* described in Section 4, is a modification that accommodates stuttering in a more natural way. In addition to the standard acceptance states, testing automata also feature *livelock accepting* states.

The work on testing automata is based on the results of [21]. There the authors defined another, slightly more complicated form of testing automaton and showed that they are more often deteministic than state-labelled Büchi automata. We extend this work in two important ways: we show how to construct our form of testing automata and provide an SCC-based algorithm for on-the-fly verification with them.

In Section 5, we compare the amount work required for on-the-fly verification using two different algorithms for transition- and state-labelled Büchi automata and our new algorithm for testing automata. It turns out that, in our experiments, the new variations were considerably more efficient in terms of the number of states and transitions they explore. An important part of the contribution of this paper comes in Section 6, where we discuss exactly how and when the differences in performance occur and attempt to explain why this is so. Our conclusions are presented in Section 7.

2 Background and Related Work

The connection between temporal logic and formal languages has been a topic of research since the 1960's [3, 23, 26]; a short but excellent overview of the development of this work and its relation to model checking is [25, Section 1.3]. The potential benefits of an automata-theoretic approach to model checking was first pointed out by Wolper in [35], and Wolper, Vardi, and Sistla in [36].

Our definitions of Kripke structures and Büchi automata are standard but, for the sake of later work, we state them explicitly. From here on we use $\mathcal{P}$ to denote a finite set of atomic propositions.

A *Kripke structure* [24] over $\mathcal{P}$ is a tuple $M = (S, I, L, R)$ where S is a finite set of states, $I \subseteq S$ is the set of initial states, $L : S \to 2^{\mathcal{P}}$ is a labelling function that maps each state s to the set of atomic propositions that are true in s, and $R \subseteq S \times S$ is the transition relation. We assume that R is total. An *execution path* or *run* of M is an infinite sequence of states $r = s_1 s_2 s_3 \ldots \in S^\omega$ such that $s_1 \in I$ and $(s_i, s_{i+1}) \in R$ for all $i \geq 1$.

A *Büchi automaton* [4] over an alphabet K is a tuple $A = (S, I, R, F)$ where S is a finite set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times 2^K \times S$ is the transition relation, and $F \subseteq S$ is a set of acceptance states. Because sets of symbols of the alphabet appear on the transitions, we shall refer to this form as a *transition-labelled Büchi automaton* (*TLBA*).

Each word accepted by A is an infinite sequence of symbols from K. A *run* of the automaton over a word $w = k_1 k_2 \ldots \in K^\omega$ is an infinite sequence of states $r = s_1 s_2 \ldots \in S^\omega$ such that $s_1 \in I$ and for all $i \geq 1$ there exists a $K_i \subseteq K$ such that $k_i \in K_i$ and $(s_i, K_i, s_{i+1}) \in R$. The set of states that occur infinitely often in run r is denoted by $inf(r)$ (and clearly $inf(r) \subseteq S$), and the run is *accepting* if and only if $inf(r) \cap F \neq \emptyset$.

When Büchi automata are used for verification we shall use $K = 2^{\mathcal{P}}$. This is interpreted in such a way that if f is a propositional logic formula over $\mathcal{P}$, and $P = \{P_1, \ldots, P_n\} \subseteq 2^{\mathcal{P}}$ is the set of all models of f (in other words, $P_i \in P$ if and only if $P_i \models f$), then we use (s, f, s') and (s, P, s') interchangeably as members of R.

2.1 Construction of Büchi Automata

An early algorithm for converting LTL formulas to Büchi automata was described by Vardi and Wolper in [34], but unfortunately it always produced automata with $2^{\mathcal{O}(n)}$ states, where n is the number of subformulas of the LTL formula. A more practical algorithm is [18], on which many later improvements are based. The basic idea is a two-step approach that first translates the input formula to a generalized Büchi automaton, which is then turned into a (standard) Büchi automaton using the flag construction due to Choueka [5].

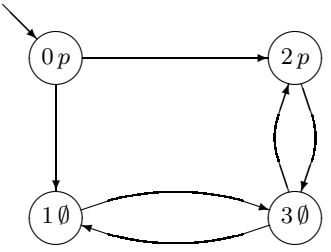
A general class of improvements is based on rewriting rules to simplify the LTL formula before any automaton is constructed, and several ad hoc heuristics have been proposed to simplify the final automaton. Several groups have proposed improvements based on different procedures for computing covering sets [9, 28], while others have concentrated on reducing the final automaton using *simulations* [12, 13, 14, 20, 29].

Gastin and Oddoux have investigated the use of *very weak alternating automata* as an intermediate form to improve both the size of the final Büchi automata and the speed of their generation [16]; this approach is not especially relevant to our work, but we shall make use of their tool for our experiments.

2.2 Verification with Büchi Automata

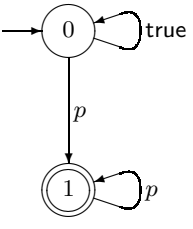
A Kripke structure M satisfies a specification ϕ if all its executions are allowed by the specification. This is equivalent to checking that none of M's executions satisfy $\neg\phi$. The automata-theoretic approach therefore consists of constructing a Büchi automaton $A_{\neg\phi}$, computing its product with M, and checking that it is empty, in other words, checking that no execution of M violates ϕ. Although it is possible to first express M itself as a Büchi automaton, the product of M and $A_{\neg\phi}$ can be defined more directly as follows.

Let $\mathcal{F} = \{\text{propositional formulas over } \mathcal{P}\}$, and let $M = (S_M, I_M, L_M, R_M)$ be a Kripke structure over $\mathcal{P}$, and $A_{\neg\phi} = (S_A, I_A, R_A, F_A)$ a TLBA over $2^{\mathcal{P}}$. Then the product of M and $A_{\neg\phi}$, denoted $M \parallel A_{\neg\phi}$, is a triple (S, R, I), where

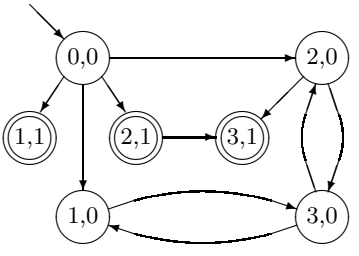


Kripke structure M

(a)

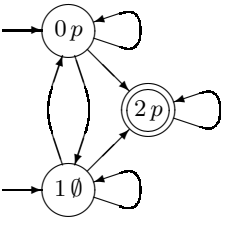


Büchi automaton $A_{\neg\phi}$

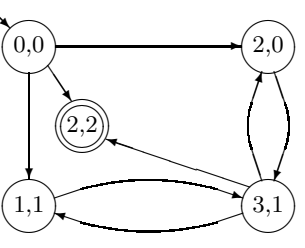


$M \parallel A_{\neg\phi}$

(b)

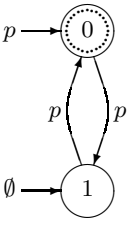


State-labelled
Büchi automaton $B_{\neg\phi}$

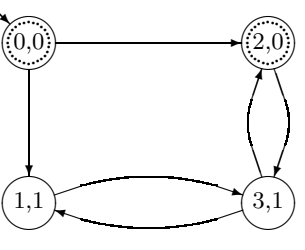


$M \parallel B_{\neg\phi}$

(c)



Testing automaton $C_{\neg\phi}$



$M \parallel C_{\neg\phi}$

Fig. 1. Examples of automata and products for verifying $\phi = \Box\Diamond\neg p$

- $S = S_M \times S_A$ is the set of states,
- $R \subseteq S \times S$ is the transition relation where $((s, a), (s', a')) \in R$ if and only if $(s, s') \in R_M \wedge \exists f \in \mathcal{F} : (a, f, a') \in R_A \wedge L_M(s) \models f$, and
- $I \subseteq I_M \times I_A$ is the set of initial states.

A run of the product is an infinite sequence of states $(s_1, a_1)(s_2, a_2) \ldots$ such that $(s_1, a_1) \in I$ and $((s_i, a_i), (s_{i+1}, a_{i+1})) \in R$ for all $i \geq 1$. A counterexample for ϕ in the product is a run such that $a_1 a_2 \ldots$ is an accepting run of $A_{\neg\phi}$.

An example of a Kripke structure, Büchi automaton, and their product is shown in Figure 1. Each state of the Kripke structure (at the top of the figure) is numbered and labelled with the set of atomic propositions that hold in the state. In this example, $\mathcal{P} = \{p\}$. The initial state is indicated by the sourceless arrow that points to the top left state. The accepting state of the Büchi automaton, shown in (a), is indicated by a double circle. The states of the product are labeled with (*Kripke state*, *Büchi state*) pairs and those state where the Büchi automaton is in an accepting state is similarly indicated by a double circle.

Arguably the most popular on-the-fly algorithm for computing the product automaton and detecting accepting cycles is a nested depth-first search algorithm first proposed by Courcoubetis, Vardi, Wolper and Yannakasis in 1990 [6]. Subsequent improvements [15, 19, 22, 27] has not only made it compatible with partial-order methods, but has also led to a significant reduction in the number of states and transitions it needs to explore. The core algorithm has also been adapted for use with generalized Büchi automata [32] and heuristic search [2, 11]. Recent work has looked again at the use of strongly connected component (SCC) algorithms for both standard and generalized Büchi automata [7, 8, 17, 27]; the algorithm we describe in Section 4.2 is based on one such.

3 State-Labelled Büchi Automata

A *state-labelled Büchi automaton* (*SLBA*) over an alphabet K is a tuple $B = (S, I, U, R, F)$ where S is a finite set of states, $I \subseteq S$ is the set of initial states, $U : S \to K$ maps each state to a symbol of the alphabet, $R \subseteq S \times S$ is the transition relation, and $F \subseteq S$ is a set of acceptance states.

A *run* of the automaton over a word $w = k_1 k_2 \ldots \in K^\omega$ is an infinite sequence of states $r = s_1 s_2 \ldots \in S^\omega$ such that $s_1 \in I$ and $(s_i, s_{i+1}) \in R$ and $U(s_i) = k_i$ for all $i \geq 1$. As for TLBAs, a run r is *accepting* if and only if $inf(r) \cap F \neq \emptyset$.

3.1 Construction of State-Labelled Büchi Automata

The conversion from a TLBA to an SLBA is straightforward. Given a TLBA $A = (S_A, I_A, R_A, F_A)$ over K, the equivalent SLBA is $B = (S_B, I_B, U_B, R_B, F_B)$ where

- $S_B = S_A \times K$, $I_B = I_A \times K$, $F_B = F_A \times K$,
- U_B maps each state to its second component, so that $U_B((s, k)) = k$, and
- R_B is such that $((s_1, k_1), (s_2, k_2)) \in R_B$ if and only if $(s_1, k, s_2) \in R_A$ for some $k \in 2^K$, and $k_2 \in k$, and k_1 is any element of K.

Some states of B may not be reachable from an initial state and can be eliminated. Isomorphic copies of subautomata of B can also be removed using an algorithm such as partition refinement. Other, more intricate optimizations are also possible but we do not focus on them here.

3.2 Verification with State-Labelled Büchi Automata

Let $M = (S_M, I_M, L_M, R_M)$ be a Kripke structure over $\mathcal{P}$, and let $B_{\neg\phi} = (S_B, I_B, U_B, R_B, F_B)$ be an SLBA over $K = 2^{\mathcal{P}}$. Then the product of M and $B_{\neg\phi}$, denoted $M \parallel B_{\neg\phi}$, is a triple (S, R, I), where

- $S = S_M \times S_B$ is the set of states,
- $R \subseteq S \times S$ is the transition relation where $((s,b),(s',b')) \in R$ if and only if $(s,s') \in R_M \wedge (b,b') \in R_B \wedge L_M(s') = U_B(b')$, and
- $I \subseteq I_M \times I_B$ are initial states where $(s,b) \in I$ if and only if $L_M(s) = U_B(b)$.

A run of the product is an infinite sequence of states $(s_1,b_1)(s_2,b_2)\ldots$ such that $(s_1,b_1) \in I$ and $((s_i,b_i),(s_{i+1},b_{i+1})) \in R$ for each $i \geq 1$. A counterexample for ϕ in the product is a run such that $b_1 b_2 \ldots$ is an accepting run of $B_{\neg\phi}$. Exactly the same algorithms used for TLBAs can be used for SLBAs.

We refer once again to Figure 1 for examples of an SLBA and its product with a Kripke structure. The notation should be clear; it corresponds to what was discussed before for the TLBA. It may seem that the difference between a TLBA and the equivalent SLBA is merely a matter of notation that carries no benefit. However, the product shown in (b) is already an early indication that this is not so: $M \parallel B_{\neg\phi}$ has two states and two transitions fewer than $M \parallel A_{\neg\phi}$.

4 Testing Automata

A *testing automaton* (*TA*) over an alphabet K is a tuple $C = (S, I, U, R, F, G)$ where S is a finite set of states, $I \subseteq S$ is the set of initial states, $U : I \to K$ maps each initial state to a symbol of the alphabet, $R \subseteq S \times K \times S$ is the transition relation, $F \subseteq S$ is a set of Büchi acceptance states, and $G \subseteq S$ is a set of livelock acceptance states.

A *run* of the testing automaton C over a word $w = k_1 k_2 \ldots \in K^\omega$ is only defined when $K = 2^{\mathcal{P}}$. In such a case, it is an infinite sequence of states $r = s_1 s_2 \ldots \in S^\omega$ such that $s_1 \in I$ and $U(s_1) = k_1$, and for all $i \geq 1$ either

1. $k_i \neq k_{i+1}$ and $(s_i, k_i \oplus k_{i+1}, s_{i+1}) \in R$, or
2. $k_i = k_{i+1}$ and $s_i = s_{i+1}$.

Here $\oplus$ denotes the symmetric difference operator on sets. A run r over a word $w = k_1 k_2 \ldots$ is *accepting* if and only if either

1. $inf(r) \cap F \neq \emptyset$ and $|inf(w)| > 1$, or
2. $\exists n : (s_n \in G) \wedge (\forall i > n : s_i = s_n \wedge k_i = k_n)$.

This general formulation of testing automata allows transitions of the form $(s, \emptyset, s')$, but since they do not add any expressive power to an automaton and are undesirable in the context of verification, we restrict our attention to automata without such transitions. However, we do not forbid them, as they are useful for the conversion algorithm outlined in the next section.

Informally speaking, a TA is an SLBA that, whenever the Kripke structure executes a stuttering transition, executes a null transition (stays in the same state). Its transitions are not labelled with propositions or formulas, but with "change sets", so that it only observes changes in atomic propositions. In addition to Büchi acceptance states, TAs also have livelock acceptance states. A run is accepted if and only if

1. it visits at least one Büchi acceptance state infinitely often and includes an infinite number of non-stuttering transitions (the $|inf(w)| > 1$ condition), or
2. it reaches a livelock acceptance state and from that point on contains only stuttering transitions.

4.1 Construction of Testing Automata

The conversion from SLBA to TA is a two-step process. Given an SLBA $B = (S_B, I_B, U_B, R_B, F_B)$ over alphabet K, we first construct an *intermediate* TA $C = (S_C, I_C, U_C, R_C, F_C, G_C)$ such that

- $S_C = S_B$, $I_C = I_B$, $F_C = F_B$, and $G_C = \emptyset$,
- $U_C(s) = U_B(s)$ for all $s \in I_C$, and
- $(s_1, k, s_2) \in R_C$ if and only if $(s_1, s_2) \in R_B$ and $k = U_B(s_1) \oplus U_B(s_2)$.

In the second step, C is converted to its final form by computing the maximal strongly stuttering-connected components, where stuttering-connected means that every state of the component can reach every other state via a sequence of zero or more transitions of the form $(s, \emptyset, s')$. Those components that are non-trivial (in other words, consists of at least two states or a single state with a self-loop) and contain at least one Büchi accepting state, are added state-by-state to the livelock acceptance states G_C. Then, every stuttering transition $(s, \emptyset, s')$ is removed. If s' is a member of I_C or G_C, we add s to the same set (and define $U_C(s) = U_C(s')$ when $s' \in I_C$). Finally we remove all unreachable states and transitions from the automaton.

Note that this construction can be carried out with any Büchi automaton, but it is only meaningful if the original property is expressible without the use of the next-state operator. It is not required, however, that the Büchi automaton itself exhibits no stuttering [21], only that the property is insensitive to stuttering. This ensures that the language accepted by the automaton remains the same.

As in the case of SLBAs, various further optimizations are possible, but we do not want to discuss them here. However, it is important to note one technical aspect that also applies to TLBAs and SLBAs, but which is especially important for TAs. The set of atomic propositions $\mathcal{P}$ may contain propositions that are never referenced by the automaton in question. For the purposes of efficient verification, such propositions should be removed from $\mathcal{P}$; they cannot influence the outcome of the verification and may lead to unnecessary work.

4.2 Verification with Testing Automata

Let $M = (S_M, I_M, L_M, R_M)$ be a Kripke structure over $\mathcal{P}$, and let $C_{\neg\phi} = (S_C, I_C, U_C, R_C, F_C, G_C)$ be a TA over $2^{\mathcal{P}}$. Then the product of M and $C_{\neg\phi}$, denoted $M \parallel C_{\neg\phi}$, is a triple (S, R, I), where

- $S = S_M \times S_C$ is the set of states,
- $R \subseteq S \times S$ is the transition relation where $((s, c), (s', c')) \in R$ if and only if either
  1. $(s, s') \in R_M \wedge (c, L_M(s) \oplus L_M(s'), c') \in R_A$, or
  2. $(s, s') \in R_M \wedge c = c' \wedge L_M(s) = L_M(s')$, and
- $I \subseteq I_M \times I_C$ are initial states where $(s, c) \in I$ if and only if $L_M(s) = U_C(c)$.

A run of the product is an infinite sequence of states $(s_1, c_1)(s_2, c_2)\ldots$ such that $(s_1, c_1) \in I$ and $((s_i, c_i), (s_{i+1}, c_{i+1})) \in R$ for each $i \geq 1$. A counterexample for ϕ in the product is a run such that $c_1 c_2 \ldots$ is an accepting run of $C_{\neg\phi}$.

As before, an example of a TA and its product with a Kripke structure can be found in Figure 1. Those states in part (c) of the picture where the TA (or the TA component of the product) is in a livelock accepting state have been marked with a dotted circle; in this particular example, the TA has no Büchi acceptance states, so that $F_C = \emptyset$ and $G_C = \{0\}$. The U_C labels are shown on the left of the TA at the source of the arrows to the initial states.

The same algorithms that are used for verification with TLBAs and SLBAs can be used with a TA to detect those violations that involve Büchi acceptance states. Also, in [21, 33] the authors propose a one-pass algorithm to detect violations involving the livelock acceptance states of the TA. Unfortunately, it is not possible to merge these into a single one-pass algorithm: while the first usually relies on a depth-first exploration of the product automaton, the key to the second algorithm is that transitions are explored in a specific, non-depth-first order. One solution is of course to first run the one algorithm, and then the other, but this is wasteful since any information that the first algorithm could conceivably gather is lost when it terminates. Moreover, a single one-pass algorithm has distinct advantages. For software model checking it is often expensive to generate transitions (which may involve steps such as garbage collection or heap canonization). Furthermore, if each state is visited only once, partial order reduction is simplified and there is no need to "remember" reductions made during a previous visit.

We now describe a new one-pass algorithm which is based on the LTL model checking algorithm in [17] (which, in turn, is based on Tarjan's algorithm for SCC detection [30]). The new algorithm detects both Büchi and livelock violations. While the algorithm works entirely reliably for Büchi violations, it does, in certain cases, fail to report an existent livelock violation. However, these circumstances are exceptional; for example, during the random experiments we present in the next section, this happened in only 2 out of 93560 (= 0.00214%) cases.

First, we review the Tarjan-based algorithm in [17], a recursive version of which called TARJAN is shown in Figure 2. The algorithm explores the product of a Kripke structure and a TLBA A (or SLBA B) and therefore does not take

```
0  for each i ∈ I do if colour[i] = WHITE then TARJAN(i)

   TARJAN(s)
1  colour[s] ← GREY
2  dfnr[s] ← low[s] ← n ; INC(n)
3  S.PUSH(s)
4  if accept[s] then A.PUSH(s)
5  for each successor t of s do
6     c ← colour[t]
7     if c = WHITE then TARJAN(s)
8     if c ≠ BLACK then UPDATE(s, t)
9  if A.TOP = s then x ← A.POP
10 if low[s] = dfnr[s] then SCC(s)

   UPDATE(s, t)
11 low[s] ← min(low[s], low[t])
12 if low[s] ≤ dfnr[A.TOP] then
13    report violation

   SCC(s)
14 repeat
15    x ← S.POP
16    colour[x] ← BLACK
17 until x = s
```

Fig. 2. The Tarjan-based algorithm presented in [17]

stuttering transitions into account. For every product state $s = (k, b)$ the Boolean predicate $accept[s]$ is true if and only if the Büchi component b is accepting; if, in other words, $b \in F_A$ (or $b \in F_B$). The algorithm is identical to Tarjan's classic algorithm, except for its use of an additional stack A where the accepting product states that appear on the depth-first search path are stored. Line 4 inserts such a state when it is first explored, and line 9 removes it once it has been fully explored. The test in lines 12 and 13 reports a violation as soon as a transition "closes" an SCC containing an accepting state. TARJAN uses colours to classify states; initially all states are unexplored and coloured WHITE. As the product automaton is explored, fully explored states are coloured BLACK, and states that are still on the depth-first stack or the component stack S, GREY. In the classic presentation of Tarjan's algorithm [1], this classification is made with Boolean flags, but it is trivial to see that the methods are equivalent.

Our new algorithm appears in Figure 3, and is called TARJAN$^+$. It operates on the product of a Kripke structure and a TA C. Given two product states $s = (k, c)$ and $s' = (k', c')$, the predicate $stutter(s, s')$ is true if and only if $c = c'$, in other words, $s \to s'$ is a stuttering transition. Predicate $accept[s]$ is true if and only if c is a Büchi acceptance state ($c \in F_C$), and predicate $livelock[s]$ is true if and only if c is livelock accepting ($c \in G_C$). The three abbreviated conditions that appear in lines 2b, 4a, and 13a are defined as follows:

$$C_1(p, s) \equiv livelock[s] \wedge (p = \bot \vee \neg stutter(p \to s))$$
$$C_2(s, t) \equiv accept[s] \wedge \neg stutter(s \to t)$$
$$C_3(s, t) \equiv livelock[s] \wedge stutter(s \to t)$$

The first change from TARJAN to TARJAN$^+$ is moving lines 4 and 9 of TARJAN into the **for**-loop in line 5; in the new algorithm the lines are labeled 4a and 9a. Although it is less efficient, this change clearly has no effect on the correctness of TARJAN. However, in the new algorithm the condition in line 4 has also been strengthened so that a Büchi acceptance state is only placed on stack A for certain transitions: it is present when the next transition explored is non-stuttering,

```
0   for each i ∈ I do if colour[i] = WHITE then TARJAN+(⊥, i)

    TARJAN+(p, s)
1   colour[s] ← GREY
2   dfnr[s] ← low[s] ← n ; INC(n)
2a  liveset[s] ← ∅
2b  if C1(p, s) then L.PUSH(s)
3   S.PUSH(s)
5   for each successor t of s do
4a    if C2(s, t) then A.PUSH(s)
6     c ← colour[t]
7     if c = WHITE then TARJAN+(s, t)
8     if c ≠ BLACK then UPDATE+(c, s, t)
9a    if A.TOP = s then x ← A.POP
9b  if L.TOP = s then x ← L.POP
9c  colour[s] ← BLUE
10  if low[s] = dfnr[s] then SCC(s)

    UPDATE+(c, s, t)
11  low[s] ← min(low[s], low[t])
12  if low[s] ≤ dfnr[A.TOP] then
13    report violation
13a if C3(s, t) then ADDLINKS(c, s, t)

    ADDLINKS(c, s, t)
18  for each u ∈ liveset[t] ∪ {t} do
19    if colour[u] ≠ GREY then
20      continue at line 18
21    if dfnr[u] ≥ dfnr[L.TOP]
22      ∧ (u ≠ t ∨ c ≠ WHITE) then
23      report violation
24    liveset[s] ← liveset[s] ∪ {u}
```

Fig. 3. The new algorithm used in this paper

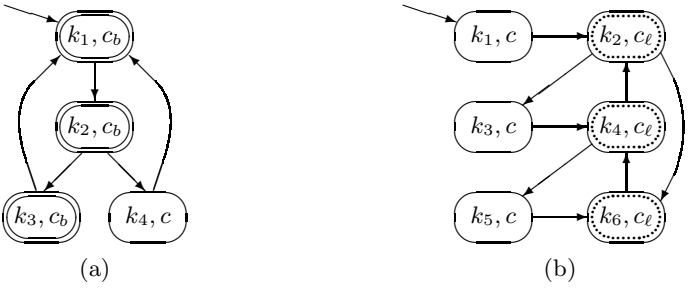


Fig. 4. Illustrative state graphs for the new algorithm

and absent when it is stuttering. This avoids the erroneous reporting of cycles that contain only stuttering transitions. Consider, for example, Figure 4(a): if $c_b \in F_C$, then states (k_1, c_b), (k_2, c_b), and (k_3, c_b) are all Büchi accepting. However, none of the states are placed on A when exploring the stuttering transitions between them, and the cycle $(k_1, c_b) \rightarrow (k_2, c_b) \rightarrow (k_3, c_b) \rightarrow (k_1, c_b)$ is therefore correctly ignored. The non-stuttering $(k_2, c_b) \rightarrow (k_4, c)$ transition satisfies C_2, and state (k_2, c_b) *is* placed on stack A before exploring it; the ensuing cycle is subsequently correctly reported as a violation.

The second change from TARJAN to TARJAN$^+$ involves the colouring of states. With the addition of line 9c, TARJAN$^+$ further distinguish those states that are currently on the depth-first stack from those that are only present in S, by colouring the latter BLUE. Once again, the detection of Büchi accepting cycles is not affected, since that part of the code (lines 7 and 8) is only concerned with the non-WHITE or non-BLACK status of states.

The last change is the introduction of the L stack and the *liveset*[ ] attribute of states. Stack L is analogous to stack A in storing the livelock acceptance states that appear on the current depth-first search path. However, an important difference is that states reached via stuttering transitions are not stored. (Only states that satisfy C_1 are pushed onto L.) For each livelock accepting state s, attribute *liveset*[s] stores the set of all other states that can be reached via already-explored stuttering transitions. For those states s' that are not livelock accepting, *liveset*[s'] $= \emptyset$. When a stuttering transition $s \rightarrow t$ is explored and state s is livelock accepting (condition C_3 in line 13a), the contents of *liveset*[t] is propagated back to s by procedure ADDLINKS. In addition, if t or some element of *liveset*[t] lies on the depth-first stack at or above the top entry of L, a livelock violation is reported (lines 21–23). (The only exception is the case where s is the direct depth-first tree parent of t, in line 22.) This is correct by the following reasoning:

1. s is livelock accepting (since $s \rightarrow t$ satisfies C_3),
2. s can reach some $u \in$ *liveset*[t] via stuttering transitions,
3. u lies at or below s on the depth-first stack (s is the top-most state on the depth-first stack and *colour*[u] = GREY), and
4. the depth-first stack transitions from u to s are stuttering, since otherwise stack L would contain an entry such that *dfnr*[u] $<$ *dfnr*[L.TOP] $\leq$ *dfnr*[s].

As mentioned above, the algorithm may in certain cases fail to detect a livelock violation. An example of this is shown in Figure 4(b). Suppose that $G_C = \{c_\ell\}$, so that (k_2, c_ℓ), (k_4, c_ℓ), and (k_6, c_ℓ) are the livelock accepting states. If the stuttering transitions $(k_4, c_\ell) \rightarrow (k_2, c_\ell)$ and $(k_2, c_\ell) \rightarrow (k_6, c_\ell)$ are explored after the non-stuttering transitions in their respective states, then the valid livelock violation $(k_2, c_\ell) \rightarrow (k_6, c_\ell) \rightarrow (k_4, c_\ell) \rightarrow (k_2, c_\ell)$ is not reported. This happens because transition $(k_6, c_\ell) \rightarrow (k_4, c_\ell)$ is explored before $(k_4, c_\ell) \rightarrow (k_2, c_\ell)$, and therefore the fact that state (k_6, c_ℓ) can reach (k_2, c_ℓ) via stuttering transitions is never recorded.

Consequently, when TARJAN$^+$ fails to report a violation, it is necessary to run the livelock detection algorithm of [21, 33] before we can claim that a Kripke structure satisfies an LTL formula, using our approach. This may appear to nullify the advantages of a single one-pass algorithm we extolled before. In practise it means that the new algorithm may be more efficient at detecting violations, but less efficient when it comes to checking that there are none.

5 Experimental Results

Table 1 shows the outcome of experiments performed to measure the effect of using SLBAs and TAs instead of TLBAs. The procedure described in [31] was used to generate 480 random 100-state Kripke structures and 360 random LTL formulas. An additional 130 formulas were taken from the literature (mostly from [13, 10, 29]), and all formulas were negated. After the elimination of stuttering-sensitive formulas and duplicates, the remaining 261 formulas were

Table 1. Comparison of automata on random graphs and random & real formulas

Automata		TLBA				SLBA				TA	
	Ave.			5.56	17.33			30.89	584.87	16.94	310.43
	Max.			53	314			389	18196	193	10944
States & transitions		TLBA				SLBA				TA	
		SE		TARJAN		SE		TARJAN		TARJAN$^+$	
All	Ave.	30.3	96.2	25.2	80.1	18.6	37.2	17.6	35.2	21.1	43.7
	Max.	3342	29404	2533	18250	1154	10184	1103	10041	1294	11284
Viol.	Ave.	33.3	105.9	27.3	85.2	20.4	33.8	19.2	31.4	20.0	31.0
	Max.	3342	29404	1652	18250	966	5888	613	3831	442	2045
Normalized		TLBA				SLBA				TA	
		SE		TARJAN		SE		TARJAN		TARJAN$^+$	
All	Ave.	51.1	19.3	38.0	15.8	33.6	8.6	32.5	8.0	57.8	11.8
	Max.	1550.0	2800.0	1250.0	2400.0	360.0	700.0	340.0	700.0	200.0	200.0
Viol.	Ave.	13.7	7.6	11.3	6.6	9.8	4.9	9.3	4.7	9.5	4.5
	Max.	1033.3	1305.3	537.5	700.0	220.0	255.1	200.0	250.0	131.1	107.5
Percentages		TLBA				SLBA				TA	
		SE		TARJAN		SE		TARJAN		TARJAN$^+$	
All	Best	0.0	0.0	3.1	1.5	0.0	0.0	4.5	2.9	10.3	11.4
	1/Best	55.7	66.5	70.2	68.5	40.5	43.4	47.9	47.8	29.0	49.9
Viol.	Best	0.0	0.0	4.1	2.0	0.0	0.0	3.7	2.5	13.7	15.1
	1/Best	56.8	60.3	65.1	62.9	23.0	26.0	30.5	30.4	38.7	38.3

converted to Büchi automata using the LTL2BA program [16], and SLBAs and TAs were constructed as described in previous sections. For TLBAs and SLBAs we used the Schwoon and Esparza modification of the CPVW algorithm [27] (shown in the "SE" column), and the Tarjan-based algorithm from [17] (shown in the "TARJAN" column). For TAs the modified Tarjan algorithm we mentioned in the previous section was used (shown in the "TARJAN$^+$" column). Even though the Kripke structures are quite small (100 states) compared to realistic models, they are large enough for our purposes. Experiments with larger Kripke structures (still random) yielded similar results.

Every cell of the table contains two numbers, the first refers to the number of states and the second to the number of transitions. The first part of the table labelled "Automata" shows the average and maximum sizes of the TLBAs, SLBAs, and TAs. From TLBA to SLBA there is roughly a 6-fold increase in the number of states and a 34-fold increase in the number of transitions. The average size of a TA is about half that of an SLBA. The next part of the table, "States & transitions" shows the average and maximum number of states and transitions explored, first in all runs, and then in only those runs where a violation was found. Unfortunately, these numbers are somewhat misleading, since large and small products carry equal weight. Therefore, the next part of the table, "Normalized", describes the same runs, but with the numbers of each run expressed as a percentage of the size of the product of the Kripke structure and the TA. The last part of the table labelled "Percentages" indicates in

what percentage of runs each automaton/algorithm pair did better than any of the others (the "Best" row), or no worse than any of the others (the "1/Best" row). Note that the figures in the TA/TARJAN$^+$ column include the number of states and transitions explored by both the TARJAN$^+$ algorithm and the livelock detection algorithm [21, 33] that is run when TARJAN$^+$ finds no violation.

We have consciously decided to report only the number of states and transitions, and not the number of bytes and milliseconds consumed by our implementations. This protects the results (to some extent) against the influence of various optimizations, implementation tricks, and the central processor and memory architecture. We generally find that the number of states gives a reliable indication of the memory required, and, similarly, the number of transitions a reliable indication of the time consumption.

When we compare only the TLBA/TARJAN and TA/TARJAN$^+$ combinations, in the case that a violation was detected, the TAs achieved a 26.7% reduction in the average number of states, and a 63.6% reduction in the average number of transitions. For the worst-case performance, TA/TARJAN$^+$ reduced the states and transitions by a factor of 3.7 and 8.9, respectively. When it comes to all runs (now including those where no violation was detected), the reduction is 16.3% and 45.4% for the average states and transitions, with factors of 2.0 and 1.6 for the worst-case states and transitions.

However, the results contain some apparent contradictions: despite the fact that the TLBA/SE combination has the highest average and worst-case numbers, it is still one of the best algorithms in more than half the cases. Conversely, the SLBA/TARJAN combination which explores the lowest average number of states in all runs, only explores the unique, least number of states in 3.7% of those runs. The cause of this phenomenon is of course the different distribution of costs for the different combinations.

It is difficult to say which of the algorithms is "best": for a single run one may use the TLBA/TARJAN combination and know that the probability is less than 0.35 that another combination can explore fewer states. Invariably, however, more than one run of a system is required and, in that case, the SLBA/TARJAN combination explores the fewest number of states and transitions, *on average*. On the other hand, the worst-case of the TA/TARJAN$^+$ combination looks more promising and it is more often the fastest ($\geq 11.4\%$ of cases) and most memory-efficient ($\geq 10.3\%$ of cases) choice.

The amount of work needed by the TLBA in the worst case is so much bigger that it tilts the averages heavily in favour of the SLBA and the TA. This, we believe, justifies the conclusion that the variants are, in fact, superior in performance to the TLBA.

Experience has shown that measurements with random Kripke structures are often over-optimistic since the "shape" of random and real state spaces can differ significantly. We would have liked to present experimental results for actual state spaces, but that approach has its own pitfalls. It is easy to find examples where one combination fares exceptionally well, while the others founder. Also, each state space should be verified against a variety of LTL properties to yield robust results. For now we have to leave this project as future work.

6 A Closer Look: Why Less Is More?

The results show that it is not so much the size of the automaton itself that counts, but rather the size of the product of Kripke structure and automaton. In the negative case—when there is no violation—all states and transitions of the product need to be explored, and, in the case of our new algorithm, it is done twice. We therefore investigated the relationship between the formulas and the size of the resulting products by devising classes of formulas of increasing length and calculating the size of the product with a set of random Kripke structures.

For the set of experiments we used 100 random 1000-state Kripke structures with a varying number of transitions. We constructed the TLBAs, SLBAs and TAs for the formulas we describe below and calculated their products with the Kripke structures. Table 2 shows the average number of states and transitions obtained in each case. The first column gives the formula class and n. The "E" formulas were of the form

$$\mathrm{E}(n) = \bigwedge_{i=1}^{n} \Diamond p_i,$$

and the "U" and "R" were of the form

$$\mathrm{U}(n) = (\cdots(p_1 \mathsf{U} p2)\mathsf{U}\cdots)\mathsf{U} p_n \quad \text{and} \quad \mathrm{R}(n) = \bigwedge_{i=1}^{n} (\Box\Diamond p_i \vee \Diamond\Box p_{i+1}).$$

The simplest, $\mathrm{E}(n)$ formulas resulted in products with the same number of states for all the automata, while the TAs produced a somewhat smaller number of transitions. The two more complicated classes result in differences that increase with the length of the formula.

Table 2. Growth of state spaces

	TLBA		SLBA		TA	
E(1)	1999.90	9126.84	1999.90	9126.84	1999.86	7730.18
E(2)	3999.68	23690.54	3999.68	23690.54	3999.62	19057.17
E(3)	7999.24	62119.04	7999.24	62119.04	7999.24	49853.79
E(4)	15998.36	164327.68	15998.36	164327.68	15998.36	134019.09
E(5)	31996.61	437872.53	31996.61	437872.53	31996.61	363967.54
U(1)	1188.75	5195.20	1145.54	5095.67	1151.85	4473.37
U(2)	3082.19	20936.82	2266.81	14020.64	2266.76	10772.02
U(3)	8702.22	92692.54	6185.00	66821.92	6233.29	47379.99
U(4)	22162.49	373415.82	15286.18	266001.99	15257.98	187339.66
U(5)	53471.09	1432869.88	35175.60	980114.68	34998.03	703055.77
R(1)	3619.32	17790.64	3326.13	16635.44	3647.73	15555.14
R(2)	9674.17	49992.48	8195.00	43328.05	8115.63	34584.78
R(3)	26620.67	150150.63	20492.62	117661.40	20113.34	88977.67
R(4)	72449.32	460101.94	51041.64	328031.95	50019.04	239408.10
R(5)	194741.06	1447741.57	127220.17	951739.06	124600.22	675327.73

We experimented with other classes as well, but were unable to find any where the TLBA products are smaller. There were classes where the results were similar to the E formulas, including

$$\mathrm{U}_2(n) = p_1 \mathsf{U} (p_2 \mathsf{U} (\cdots p_{n-1} \mathsf{U} p_n) \cdots),$$

$$\mathrm{C}_1(n) = \bigvee_{i=1}^{n} \Box\Diamond p_i \quad \text{and} \quad \mathrm{C}_2(n) = \bigwedge_{i=1}^{n} \Box\Diamond p_i.$$

Other classes performed much like the U and R formulas, for example

$$\mathrm{Q}(n) = \bigwedge_{i=1}^{n} (\Diamond p_i \vee \Box p_{i+1}) \quad \text{and} \quad \mathrm{S}(n) = \bigvee_{i=1}^{n} \Box p_i.$$

The smaller products have very little, if anything, to do with the livelock acceptance states of the testing automata, since the state-labelled automata result in products that are just as small. So the question remains, why do the SLBAs and TAs produce smaller products? We cannot give a definitive answer, but we believe that there are two important factors:

Firstly, an SLBA makes a finer distinction between different states, in the sense that the state of the SLBA contains more information about the state of the product than is the case for the TLBA. Undoubtably the TLBA is a more dense representation of the property than the equivalent SLBA. In other words, in the product several Kripke states may be paired with the same TLBA state, but because the automaton will later have to distinguish between the states, extra work needs to be performed.

Secondly, and partly because of the first reason, the TLBA is, in a sense, more nondeterministic and therefore, on average, more of its transitions are enabled in a given state. In [28] the authors suggest that more deterministic, rather than smaller automata result in smaller products, and to some extent, the generation of the SLBA removes some of the nondeterminism.

The sometimes significantly smaller number of transitions in the products of TAs can be explained, at least in part, by the fact that they have no stuttering transitions and therefore cannot cause a multiplication of stuttering steps of the Kripke structure. The theoretical results in [21], which state that testing automata are more often deterministic, do not, however explain anything at all in these findings, since the SLBA and TA products have almost exactly the same number of states.

The size of the product is not an accurate measure of performance when there actually is a violation. It might be the case that a counterexample is found relatively early on in a bigger product. This may be due to two factors. Firstly, the decision of which transitions to explore first in the on-the-fly algorithm may play a crucial role. An endless variety of heuristics and shufflings of transitions are possible and we currently know of no definitive way to decide which is best. Secondly, the counterexamples themselves may have different properties. The SLBA- or TA-induced product may be smaller but contain only relatively few,

lengthy and complicated counterexamples, whereas the product arising from a TLBA may be big but have more shallow and simple counterexamples. One open question is exactly what the relative contribution of the two phenomena in different circumstances is.

7 Conclusions

We have investigated two alternatives to the standard (transition-labelled) form of Büchi automata, namely state-labelled Büchi automata and testing automata, described the conversion from the standard form to the variant forms, and sketched our current (SCC-based) algorithm for verification with testing automata. Even though the differences between the automata may appear to be merely a matter of notation, our experimental results suggest that there are real benefits to be had from using the variant forms.

To explain the improved performance of the variants, we considered simple classes of LTL_{-X} formulas and compared the products of a set of random Kripke structures with the transition-labelled and state-labelled Büchi automata and testing automata. Despite the fact that the variant automata are invariably much larger, the resulting product automata are invariably smaller. In the case of testing automata, the number of transitions is clearly smaller and grows at a much slower rate than is the case for the other two automata.

Two factors that play a role in this phenomenon are (1) that SLBAs and TAs make finer distinctions among states, and (2) that they are more often deterministic than standard Büchi automata. This concurs with the work in [28], where the authors focused on the standard form.

Our research perhaps raises more questions than it answers. By no means do we wish to discourage further work on the reduction of Büchi automata or other ω-automata; rather our results point to the need to further investigate the factors that lead to improved performance. Other lines of future research include the characterization of LTL_{-X} properties for which the SLBAs and TAs do better, and an extension of these results to generalized Büchi automata and alternating automata.

Acknowledgments. The work of H. Hansen was supported by the Nokia Foundation.

References

1. A. V. Aho, J. E. Hopcroft, & J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. L. Brim, I. Černá, M. Nečesal. Randomization helps in LTL model checking. In *Proc. Joint Intl. Worksh. Process Alg. and Probabilistic Methods, Performance Modeling and Verif.*, LNCS #2165, pp. 105–119, Sept 2001.
3. J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Math.* 6, pp. 66–92, 1960.

4. J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. 1960 Intl. Congr. Logic, Method and Philosophy of Science*, pp. 1–11, Stanford Univ. Press, Jun 1962.
5. Y. Choueka. Theories of automata on ω-tapes: a simplified approach. *Journal Computer and System Sciences* 8, pp. 117–141, 1974.
6. C. Courcoubetis, M. Y. Vardi, P. Wolper, M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *CAV'90*, LNCS #531, pp. 233–242, Jun 1990. Journal version: *Formal Methods in System Design* 1(2/3), pp. 275–288, Oct 1992.
7. J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proc. World Congr. Formal Methods in the Development of Computing Systems (FM'99)*, LNCS #1708, pp. 253–271, Sept 1999.
8. J.-M. Couvreur. On-the-fly emptiness checks for generalized Büchi automata. In *Proc. 12th Intl.* SPIN *Worksh. on Model Checking Software*, LNCS #1708, pp. 999–999, Aug 2005.
9. M. Daniele, F. Giunchiglia, M. Y. Vardi. Improved automata generation for linear time temporal logic. In *CAV'99*, LNCS #1633, pp. 249–260, Jul 1999.
10. M. B. Dwyer, G. S. Avrunin, & J. C. Corbett. Property specification patterns for finite-state verification. In *Proc. 2nd ACM Worksh. Formal Methods in Software Practice*, pp. 7–15, Mar 1998.
11. S. Edelkamp, S. Leue, A. Lluch Lafuente. Directed explicit-state model checking in the validation of communication protocols. Technical Report 161, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Oct 2001.
12. K. Etessami. A hierarchy of polinomial-time computable simulations for automata. In *CONCUR'02*, LNCS #2421, pp. 131–144, Aug 2002.
13. K. Etessami, G. J. Holzmann. Optimizing Büchi automata. In *CONCUR'00*, LNCS #1877, pp. 154–167, Aug 2000.
14. C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In *Proc. 8th Intl. Conf. Implementations and Application of Automata*, LNCS #2759, pp. 35–48, Jul 2003.
15. P. Gastin, P. Moro, M. Zeitoun. Minimization of counterexamples in SPIN. In *Proc. 11th Intl.* SPIN *Worksh. Model Checking Software*, LNCS #2989, pp. 92–108, Apr 2004.
16. P. Gastin, D. Oddoux. Fast LTL to Büchi automata translation. In *CAV'01*, LNCS #2102, pp. 53–65, Jul 2001.
17. J. Geldenhuys, A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *TACAS'04*, LNCS #2988, pp. 205–219, Mar–Apr 2004. Journal version: *Theor. Computer Science* 345(1), pages 60-82, Nov 2005.
18. R. Gerth, D. Peled, M. Y. Vardi, P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th IFIP Symp. Protocol Spec., Testing, and Verif.*, pp. 3–18, Jun 1995.
19. P. Godefroid, G. J. Holzmann. On the verification of temporal properties. In *Proc. 13th IFIP Symp. Protocol Spec., Testing, and Verif.*, pp. 109–124, May 1993.
20. S. Gurumurthy, R. Bloem, F. Somenzi. Fair simulation minimization. In *CAV'02*, LNCS #2404, pp. 610–624, Jul 2004.
21. H. Hansen, W. Penczek, A. Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. In *Proc. 7th Intl. ERCIM Worksh. Formal Methods for Industrial Critical Systems*, pp. 185–200, Jul 2002. Also published in Elec. Notes in Theor. Computer Science 66(2), Elsevier Science, Dec 2002.
22. G. J. Holzmann, D. Peled, M. Yannakakis. On nested depth first search. In *Proc. 2nd* SPIN *Worksh.*, Held Aug 1996, DIMACS Series No. 32, pp. 23–32, 1997.

23. J. A. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, Univ. of California, 1968.
24. S. A. Kripke. Semantical analysis of modal logic I, normal propositional calculi. *Zeitschrift für mathematische Logik und Grundlagen der Math* 9, pp. 67–96, 1963.
25. R. P. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-theoretic Approach*. Princeton Univ. Press, 1994.
26. M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. of the American Mathemathical Society* 141, pp. 1–35, 1969.
27. S. Schwoon, J. Esparza. A note on on-the-fly verification algorithms. In *TACAS'05*, LNCS #3440, pp. 174–190, Mar 2005.
28. R. Sebastiani, S. Tonetta. "More Deterministic" vs "Smaller" Büchi Automata for Efficient LTL Model Checking *Correct Hardware Design and Verif. Methods*, LNCS #2860, pp. 126–140, 2003.
29. F. Somenzi, R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV'00*, LNCS #1855, pp. 248–267, Jun 2000.
30. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing* 1(2), pp. 146–160, Jun 1972.
31. H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulae In *Proc. Worksh. Concurrency, Specifications, and Programming*, pp. 251–262, Sept 1999.
32. H. Tauriainen. Nested emptiness search for generalized Büchi automata. Technical Report HUT–TCS–A79, Laboratory for Theoretical Computer Science, Helsinki Univ. of Technology, Jul 2003.
33. A. Valmari. On-the-fly verification with stubborn sets. In *CAV'93*, LNCS #697, pp. 397–308, Jun 1993.
34. M. Y. Vardi, P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symp. on Logic in Computer Science*, pp. 332–344, Jun 1986.
35. P. Wolper. Temporal logic can be more expressive. *Information and Computation* 56, pp. 72–99, 1983.
36. P. Wolper, M. Y. Vardi, A. P. Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symp. on the Foundations of Computer Science*, pp. 185–194, IEEE Computer Society Press, Nov 1983.