On-the-Fly Emptiness Checks for Generalized Büchi Automata

Jean-Michel Couvreur[1], Alexandre Duret-Lutz[2], and Denis Poitrenaud[2]

[1] LaBRI, Université de Bordeaux I, Talence, France
[2] LIP6, Université de Paris 6, France

Abstract. *Emptiness check* is a key operation in the automata-theoretic approach to LTL verification. However, it is usually done on Büchi automata with a single acceptance condition. We review existing on-the-fly emptiness-check algorithms for *generalized* Büchi automata (i.e., with *multiple acceptance conditions*) and show how they compete favorably with emptiness-checks for degeneralized automata, especially in presence of weak fairness assumptions. We also introduce a new emptiness-check algorithm, some heuristics to improve existing checks, and propose algorithms to compute accepting runs in the case of multiple acceptance conditions.

1 Introduction

The automata-theoretic approach to model-checking [22] uses automata on infinite words to represent a system as well as a formula to check on this system. Both automata are synchronized, and a key operation is to determine whether the resulting automaton is empty (i.e., contains no accepting run). This operation is called *emptiness check*. An on-the-fly emptiness check allows the synchronized automata to be constructed lazily while it runs. This is a win if the emptiness check answers before the whole synchronized product is completed.

We follow up on a paper by Schwoon and Esparza [17] who compared two classes of on-the-fly emptiness checks: those based on nested depth-first searches (NDFSs) versus those computing strongly connected components (SCCs). Their measures for Büchi automata with single acceptance conditions led to the following conclusions:

- Couvreur [3]'s algorithm is the best at computing accepting SCCs,
- Schwoon and Esparza [17]'s algorithm is the best of NDFS-based checks,
- for weak Büchi automata [1], a simple DFS is enough; otherwise SCC-based algorithms should be preferred to NDFSs unless bit-state hashing is used.

Here we explore these algorithms on Büchi automata with multiple acceptance conditions (the so-called *generalized Büchi automata*) to stress the advantages of generalized emptiness checks over traditional algorithms.

Section 2 introduces the emptiness-check problem and existing algorithms. Section 3 describes our experimental workbench. The later two sections present some contributions to each class of algorithms as well as algorithms for the computation of accepting runs.

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 169–184, 2005.

2 Emptiness Check

2.1 Transition-Based Generalized Büchi Automata

A *Transition-based Generalized Büchi Automaton* (TGBA) over the alphabet Σ is a Büchi automaton with labels on transitions, and generalized acceptance conditions on transitions too. It can be defined as a tuple $A = \langle \Sigma, \mathcal{Q}, \mathcal{F}, q^0, \delta \rangle$ where

- Σ is an alphabet,
- $\mathcal{Q}$ is a finite set of elements called *states*,
- $\mathcal{F}$ is a finite set of elements called *acceptance conditions*,
- $q^0 \in \mathcal{Q}$ is a distinguished initial state,
- $\delta \subseteq \mathcal{Q} \times (2^\Sigma \setminus \{\emptyset\}) \times 2^\mathcal{F} \times \mathcal{Q}$ is the transition relation, where each transition is labeled by a nonempty set of letters of Σ and a set of acceptance conditions of $\mathcal{F}$.

A *run* of A is an infinite sequence $\langle q_0, l_0, f_0, q_1 \rangle \langle q_1, l_1, f_1, q_2 \rangle \cdots \langle q_j, l_j, f_j, q_{j+1} \rangle \cdots$ of transitions of δ, starting at $q_0 = q^0$. Such a run is said to be *accepting* if $\forall f \in \mathcal{F}$, $\forall i \geqslant 0$, $\exists j \geqslant i$, such that $f \in f_j$, i.e., if its transitions are labeled by each acceptance condition infinitely often.

An *emptiness check* is an algorithm that tells whether at least one accepting run exists. On a TGBA, it amounts to testing whether there exists a circuit that (1) is accessible from q^0, and (2) is labeled by all acceptance conditions $\mathcal{F}$.

The l_is can be used to describe words recognized by a run, but for the purpose of finding accepting runs we shall not be concerned by l_is and Σ. Also note that we use acceptance conditions as labels on transitions, rather than as the usual sets of transitions, because that is how it is coded in practice.

Other Büchi automata in use for model-checking, have acceptance conditions on states rather than transitions, and are often not generalized (i.e., $|\mathcal{F}| \leqslant 1$). While the benefit of TGBAs in the process of translating LTL formulæ is already quite clear [3, 10, 4], few people are actually using them for emptiness-check, because mainstream algorithms work on non-generalized, state-based, Büchi automata.

A *degeneralization* is the transformation of an automaton with $|\mathcal{F}| > 1$ into an automaton with $|\mathcal{F}| = 1$ [10]. This operation may multiply the size of the automaton by at most $|\mathcal{F}|$ to produce a transition-based automaton, and by at most $|\mathcal{F}| + 1$ to produce a state-based automaton. Such a blowup is often disregarded when only the automaton that represents the property needs to be degeneralized: such automata are usually small. Acceptance conditions can also be used to express some class of fairness constraints such as *weak fairness*. In Spin, weak fairness is handled using a degeneralization algorithm [13, p. 182]. As we shall see in our measures, the degeneralization is much more painful when applied to weak fairness.

2.2 Existing Algorithms

Two classes of on-the-fly emptiness-check algorithms exist: nested depth-first searches (NDFSs), and algorithms that compute strongly connected components (SCCs). Fig. 1 shows how the algorithms we cite relate to each other.

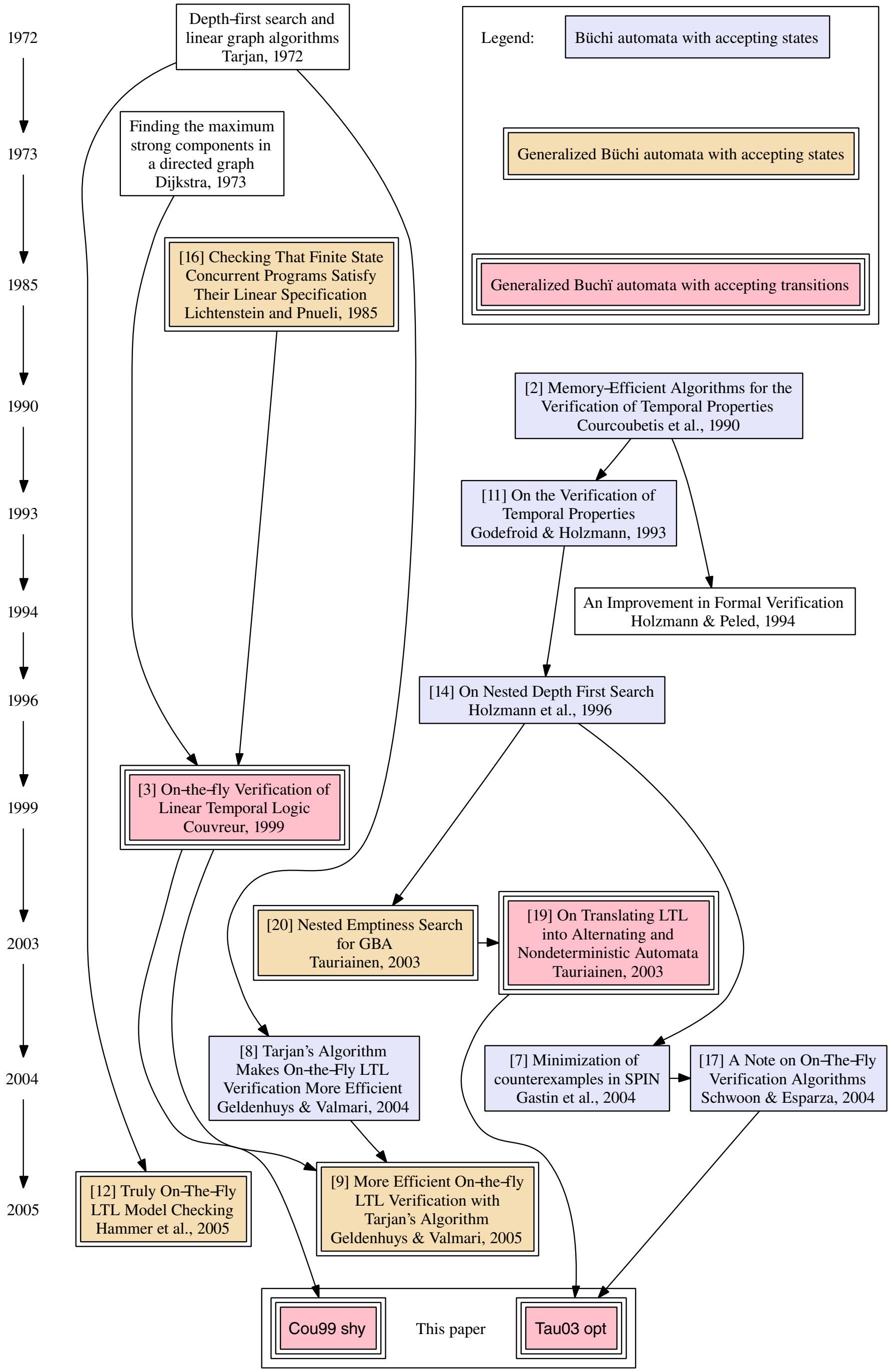
1972
1973
1985
1990
1993
1994
1996
1999
2003
2004
2005
Depth–first search and linear graph algorithms Tarjan, 1972
Legend:
Büchi automata with accepting states
Generalized Büchi automata with accepting states
Generalized Buchï automata with accepting transitions
Finding the maximum strong components in a directed graph Dijkstra, 1973
[16] Checking That Finite State Concurrent Programs Satisfy Their Linear Specification Lichtenstein and Pnueli, 1985
[2] Memory–Efficient Algorithms for the Verification of Temporal Properties Courcoubetis et al., 1990
[11] On the Verification of Temporal Properties Godefroid & Holzmann, 1993
An Improvement in Formal Verification Holzmann & Peled, 1994
[14] On Nested Depth First Search Holzmann et al., 1996
[3] On–the–fly Verification of Linear Temporal Logic Couvreur, 1999
[20] Nested Emptiness Search for GBA Tauriainen, 2003
[19] On Translating LTL into Alternating and Nondeterministic Automata Tauriainen, 2003
[8] Tarjan's Algorithm Makes On–the–Fly LTL Verification More Efficient Geldenhuys & Valmari, 2004
[7] Minimization of counterexamples in SPIN Gastin et al., 2004
[17] A Note on On–The–Fly Verification Algorithms Schwoon & Esparza, 2004
[12] Truly On–The–Fly LTL Model Checking Hammer et al., 2005
[9] More Efficient On–the–fly LTL Verification with Tarjan's Algorithm Geldenhuys & Valmari, 2005
Cou99 shy
This paper
Tau03 opt

Fig. 1. A family tree of emptiness-check algorithms

```
// Let $\langle \Sigma, \mathcal{Q}, \delta, q^0, \mathcal{F} \rangle$ be the
// input automaton to check.
$todo$: stack of $\langle state \in \mathcal{Q}, succ \subseteq \delta \rangle$
$SCC$: stack of $\langle root \in \mathbb{N}, la \subseteq \mathcal{F},$
                $acc \subseteq \mathcal{F}, rem \subseteq \mathcal{Q} \rangle$
$H$: map of $\mathcal{Q} \mapsto \mathbb{N}$
$max \leftarrow 0$

main():
  push($\emptyset$, $q^0$)
  while $\neg todo$.empty()
    if $todo$.top().$succ = \emptyset$
      pop()
    else
      pick one $\langle \_, \_, a, d \rangle$ off $todo$.top().$succ$
      if $d \notin H$
        push($a$, $d$)
      else if $H[d] > 0$
        if merge($a$, $H[d]$) $= \mathcal{F}$
          return $\bot$
  return $\top$

push($a \subseteq \mathcal{F}$, $q \in \mathcal{Q}$):
  $max \leftarrow max + 1$
  $H[q] \leftarrow max$
  $SCC$.push($\langle max, a, \emptyset, \emptyset \rangle$)
  $todo$.push($\langle q, \{\langle s, l, a, d \rangle \in \delta \mid s = q\} \rangle$)

pop():
  $\langle q, \_ \rangle \leftarrow todo$.pop()
  $SCC$.top().$rem$.insert($q$)
  if $H[q] = SCC$.top().$root$
    forall $s \in SCC$.top().$rem$
      $H[s] \leftarrow 0$
    $SCC$.pop()

merge($a \subseteq \mathcal{F}$, $t \in \mathbb{N}$):
  $r \leftarrow \emptyset$
  while ($t < SCC$.top().$root$)
    $a \leftarrow (a \cup SCC$.top().$acc$
            $\cup\, SCC$.top().$la)$
    $r \leftarrow r \cup SCC$.top().$rem$
    $SCC$.pop()
  $SCC$.top().$acc \leftarrow SCC$.top().$acc \cup a$
  $SCC$.top().$rem \leftarrow SCC$.top().$rem \cup r$
  return $SCC$.top().$acc$
```

Fig. 2. Another presentation of the algorithm of Couvreur [3] to check the emptiness of TGBAs

Nested Depth-First Searches. NDFSs were initially developed for Büchi automata with only one acceptance condition for states [2]. Basically, a NDFS will perform a first DFS rooted at q^0 until it finds an accepting state s, and from there starts a second DFS to check whether s is reachable from itself. This naive algorithm was then further refined so that both DFSs could share the same hash table [11], to exit earlier and to support partial order reductions [14].

Holzmann et al. [14]'s algorithm has been refined by Gastin et al. [7] and Schwoon and Esparza [17]. In parallel, Tauriainen generalized it to support multiple acceptance conditions on states [20], or transitions [19]. Switching from states to transitions is easy; the real challenge was to devise a way to handle generalized acceptance conditions. Tauriainen did this by repeating the inner DFS several times (at worst $|\mathcal{F}|$ times).

Strongly Connected Components. Another strategy is to compute the maximal strongly connected components (MSCCs) of the automaton. Let us define a *trivial* SCC as a single state without self-loop. If the union of all the acceptance conditions occurring in a non-trivial SCC is $\mathcal{F}$, and that SCC is accessible from q^0, then one can assert the existence of such an accepting run. This is the essence of the algorithms of Couvreur [3], Geldenhuys and Valmari [8, 9], and Hammer et al. [12].

We present an iterative version of Couvreur [3]'s algorithm in Fig. 2 in order to introduce two heuristics in Section 4.1. This algorithm is based on the fact that any

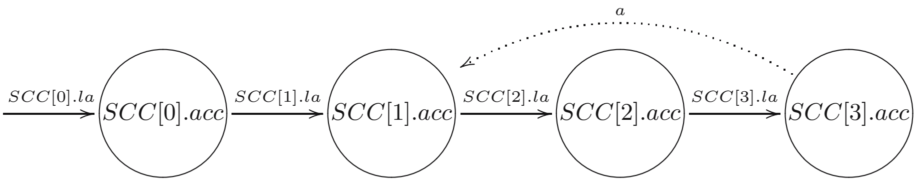

Fig. 3. The meaning of la and acc in SCC

graph contains at least one MSCC without outgoing arc. To list all MSCCs, one should find such a terminal MSCC, remove it from the graph, and then list all MSCCs of the resulting graph [16]. It turns out this requires to visit each transition only once.

To do so the algorithm explores the graph in depth-first order. $todo$ is a DFS stack, on which each item contains a state and the set of its successors that have yet to be visited. (In practice this set of successors may not need to be represented explicitly and would be replaced by the necessary information to compute the next successor of the state.) H maps each state to its rank in the depth-first order, and $H[q] = 0$ indicates that q belongs to a removed MSCC.

During the DFS, a chain of SCCs is maintained as a stack, SCC, depicted on Fig. 3. To each SCC is associated the rank of the first state of the SCC ($root$), the union of acceptance conditions in the SCC (acc), the acceptance conditions labeling the transition coming from the previous SCC (la), and the list of states of the SCC that have been fully explored (rem). ($SCC[0].la = \emptyset$ by convention and is never used.) Using this structure, two visited states q_1 and q_2 belong to the same SCC if $\max\{r \mid SCC[r].root \leqslant H[q_1]\} = \max\{r \mid SCC[r].root \leqslant H[q_2]\}$.

Initially, each new state is pushed on the stack as a trivial SCC with an empty acc (line c26). When the DFS reaches a successor q that has already been visited and has not been removed (line c18), all SCCs between the SCC to which q belongs and the top SCC (source of the transition) are merged into a single SCC. On the example of Fig. 3 where a back arc is found between $SCC[3]$ and $SCC[1]$, the last three SCCs would be merged into a single one with acceptance conditions $SCC[1].acc \cup SCC[2].la \cup SCC[2].acc \cup SCC[3].la \cup SCC[3].acc \cup a$. If that union is $\mathcal{F}$, then an accessible, non-trivial, and accepting SCC exists, and the algorithm reports $\bot$ (the automaton is not empty).

When the root of an SCC is popped (tested line c32), the SCC is known to be maximal and not accepting, so it can be discarded. The use of rem line c31 to remove the states of the MSCC line c33 could be avoided because when line c33 is reached, rem contains all the states s accessible from q (ignoring those with $H[s] = 0$), as the original algorithm did [3]. The current implementation favors run-time to memory consumption, indeed a concern from Schwoon and Esparza [17] was that computing transitions can be expensive. To be fair we will account for the size of rem in our measures of the stack size. (Geldenhuys and Valmari [9] provide alternative structures that address the same problem.)

Another SCC-based algorithm, Geldenhuys and Valmari [8]'s, has a similar handling of its stack: it keeps all states of partial SCCs, so it can remove them easily.

However it also stores an additional integer for each state ($lowlink$) that we will not account for in our measures. This algorithm works only on degeneralized automata.

Hammer et al. [12]'s algorithm is presented as an emptiness check for Linear Weak Alternating Automata (LWAA). However their algorithm translates an LWAA into a generalized Büchi automata on-the-fly during the emptiness check. The translation from LWAA could be coupled with any other emptiness-check algorithm presented here. The real part of their emptiness check follows the same logic as Couvreur's algorithm except it merges SCCs one by one while popping instead of immediately when a loop is found. It will therefore find an accepting SCC later than the algorithm of Fig. 2, only when this SCC is popped.

3 Experimentations

In this section we introduce the experimental framework in which we compare the aforementioned algorithms, and comment on the results. All the algorithms we use are implemented in the Spot library [4]. The random graph and random LTL formulæ generation algorithms are comparable to those presented by Tauriainen [21]. Of the 8 emptiness-check algorithms we compare, the first 4 are SCC-based: Cou99, is the algorithm of Fig. 2, Cou99 shy- and Cou99 shy are two variants of Cou99 described in Section 4.1, and GV04 is the algorithm of Geldenhuys and Valmari [8]. The other 4 are NDFS algorithms: CVWY90 [2], SE05 [17], Tau03 [19], and Tau03 opt (a variant of Tau03 presented Section 4.2). In tables, "×" indicates new algorithms that will be discussed in Section 4.

Because all our tests use TGBAs as input, we had to adjust CVWY90, GV04, and SE05 to handle transition-based automata (this is straightforward) and because they will not handle generalized acceptance conditions we also had to degeneralize the input automata for these 3 algorithms. (Hence the input can be $|\mathcal{F}|$ times larger.)

We exercised these algorithms on random graphs and concrete models, following a pattern similar to that of Geldenhuys and Valmari [8]. First we use them to check random graphs against LTL formulæ. Then we try them on two real models (the first of which also comes from Geldenhuys and Valmari [8]).

Table 1 presents our results when checking random graphs with all algorithms in 12 different setups. Each setup differs in how the graph and formulæ are generated. The random graphs have 1024 states and are generated with 3 different densities d of transitions (all 1024 states are accessible and the arity of each state follows a normal distribution with mean $1+1023d$ and variance $1023d(1-d)$). In columns headed "fair", transitions in the graph are additionally randomly labeled with 3 acceptance conditions to mimic weak fairness constraints; in the other columns the acceptance conditions, if any, will come only from the LTL formulæ.

The values presented for each experiment are means. They where computed by running each emptiness check on a set of 1300–3000 products generated as follows.

For each setup we consider 15 random graphs. On setups with random formulæ, each graph is checked against 200 LTL formulæ (converted to TGBAs using the algorithm of Couvreur [3]), yielding 3000 different products. On setups with "Human-generated formulæ", each graph is checked against 94 formulæ (and their negation) selected from the literature [5, 6, 18], yielding 2820 products.

In this first test we discarded all products without accepting run (i.e., keeping those where algorithms need not visit the whole automaton). For each setup the number of non-empty products remaining is written in italics on the same line as the density. For Tau03 we also discarded products with no acceptance conditions at all, because Tau03 is not designed to handle them; the resulting number of products is put in parentheses.

For each check of a non-empty product, we compute the ratios between (1) the number of distinct states visited and the number of states in the product TGBA, (2) the number of traversed transitions (a same transition can be accounted more than once) and the number of transitions of the product TGBA, and (3) the maximal size of the stack and the number of states of the product. For all algorithms, even if a degeneralization is required, ratios are computed against the product before any degeneralization. The table displays the means of each of these three ratios in %.

Our computation of the stack size deserves more explanations as not all algorithms use similar stacks. For all NDFS algorithms, we simply counted the number of states in the DFS stack. For Cou99, we counted the number of entries in $todo$ (its DFS stack), plus the size of rem for each of entry on SCC (this is because $succ$ can be represented as an iterator of constant size, and if you omit its rem field the size of SCC is bounded by that of $todo$). For GV04 we counted all items on $stack$ [8] (this is proportional to all states that are in the current SCC chain).

The algorithms presented here have a runtime proportional to the number of transitions explored. So the second value of each triplet allows to compare the runtimes. Also, for a fixed $|\mathcal{F}|$, the memory consumption of the algorithm is a linear combination of the number of states explored (first value) and of the size of the stack (second value).

A first remark concerns the results presented by Geldenhuys and Valmari [8], who compared their implementations of GV04 and CVWY90 using the same procedure (at the exception of the "fair" columns). We could not reproduce the important contrast they show between these two algorithms (neither could Hammer et al. [12]). For example, the 94 formulæ (and their negation) from the literature have been checked against 15 random graphs with a transition probability of 0.001. 2308 of the 2820 generated products are non-empty and GV04 has reported an accepting run after exploring an average of 7.5% of the states when CVWY90 needs to visit 7.7%. Geldenhuys and Valmari [8] report a rate of 8.99% for GV04 against 40.21% for CVWY90. These discrepancies are likely due to different parameters of the random graph generator.

The results for setups with randomly-generated formulæ are comparable to those with non-random formulæ; if anything, this only shows that random formulæ are not biased. Similarly, the density d does not seem to affect the algorithms much. Therefore it is much more interesting to compare the behaviors of the algorithms when acceptance conditions comes only from the formulæ or when additional acceptance conditions have been injected into the random graphs. In the former most TGBAs have few acceptance conditions (e.g., for formulæ from the literature, 40% of TGBAs have 0 or 1 acceptance condition, 40% have 2, and 20% have between 3 and 6), consequently the difference between CVWY90, GV04, SE05 (which require degeneralized automata) and the other algorithms are not striking. However on the "fair" setups, SCC-based algorithms often outrank NDFS ones. The poor results of Tau03 are mostly due to the logic of the origi-

Table 1. Comparison of algorithms for random graphs and random and real LTL formulæ

		Random formulæ						Human-generated formulæ						
degeneralized	Algorithm	formula's cond.			fair			formula's cond.			fair			
	$d = 0.001$	*2328 (1318)*					*2188*	*2308 (2127)*					*1951*	
	Cou99	6.8	4.5	4.5	18.1	11.1	13.8	7.4	5.1	4.0	16.3	10.6	10.4	SCC-based
×	Cou99 shy-	5.4	5.8	7.5	16.5	17.7	25.6	6.5	6.8	7.7	15.2	15.7	19.6	
×	Cou99 shy	5.4	5.8	7.4	15.6	16.7	23.5	6.3	6.5	7.0	14.5	15.0	18.0	
	GV04	6.8	4.5	4.5	28.4	17.1	21.6	7.5	5.1	4.0	25.9	16.5	16.1	
	CVWY90	6.8	7.1	6.4	61.7	73.9	66.6	7.7	7.9	5.2	53.6	65.0	49.3	NDFS
	SE05	6.8	5.7	4.5	59.4	39.1	38.4	7.6	6.8	3.9	50.9	34.7	28.1	
	Tau03	9.9	16.1	10.8	64.7	295.9	49.6	9.5	17.4	8.1	53.9	265.5	36.2	
×	Tau03 opt	6.8	5.2	4.5	18.5	27.1	15.4	7.4	8.1	3.8	16.4	31.8	11.3	
	$d = 0.002$	*2716 (1488)*					*2695*	*2569 (2304)*					*2548*	
	Cou99	4.8	2.2	3.1	17.5	8.5	11.4	4.5	2.6	2.4	13.7	7.4	7.6	
×	Cou99 shy-	3.4	3.4	6.9	15.4	15.8	30.1	3.6	3.8	6.1	12.5	12.1	19.8	
×	Cou99 shy	3.4	3.4	6.8	14.3	14.6	26.5	3.4	3.6	5.4	11.9	11.4	17.3	
	GV04	4.8	2.2	3.1	29.1	14.1	18.5	4.6	2.8	2.4	23.2	12.8	11.9	
	CVWY90	4.9	3.6	4.5	60.3	58.0	59.5	4.8	4.2	3.4	45.1	43.9	35.6	
	SE05	4.8	2.8	3.1	56.8	30.2	32.9	4.7	3.5	2.4	42.0	24.3	19.8	
	Tau03	8.5	12.5	9.1	61.3	265.5	46.5	7.1	12.5	6.3	40.9	185.4	27.3	
×	Tau03 opt	4.8	2.7	3.1	17.8	23.9	12.7	4.5	4.5	2.4	13.9	26.3	8.3	
	$d = 0.01$	*2978 (1569)*					*2979*	*2766 (2441)*					*2765*	
	Cou99	3.5	0.7	2.4	12.3	1.9	8.1	2.6	0.7	1.4	7.8	1.5	4.8	
×	Cou99 shy-	1.7	1.5	11.7	8.2	8.3	66.6	1.6	1.6	10.6	5.6	5.4	39.2	
×	Cou99 shy	1.6	1.5	10.7	6.9	7.0	53.0	1.4	1.3	7.7	4.8	4.4	29.9	
	GV04	3.5	0.7	2.4	20.7	3.5	13.2	2.6	0.8	1.4	13.5	2.8	7.7	
	CVWY90	3.6	1.1	3.3	44.7	15.6	49.8	2.7	1.0	2.1	30.8	13.0	30.9	
	SE05	3.6	0.9	2.3	39.8	7.0	25.4	2.6	0.9	1.5	26.4	5.7	15.5	
	Tau03	16.9	20.6	19.7	58.1	221.8	57.8	11.2	14.9	12.7	35.5	140.2	32.3	
×	Tau03 opt	3.5	1.0	2.3	12.4	11.0	9.6	2.6	1.6	1.4	7.8	10.2	5.7	

nal algorithm [19]; informally, it visits all the successors of a state even if it could have answered after having visited the first.

Experiments based only on random graphs can be misleading. To emphasize the advantage of TGBAs and SCC-based emptiness checks, we have verified concrete formulæ against concrete models. For this purpose, we have treated one example presented by Geldenhuys and Valmari [8] modeling an algorithm of election in an arbitrary network (this model is also experimented by Schwoon and Esparza [17]). Among the three variations they presented [8], Table 2 collects our results only for the second one, checked against their 9 formulæ (labeled from A to I). Values for the other, less significant variations can be computed using the benchmark scripts distributed with Spot.

Each square corresponds to a given formula. At the top of a square is indicated the label of the formula as well as the product size (in terms of number of states, transitions and acceptance conditions). Moreover, a symbol indicates if the product is empty (∅) or if an accepting run exists (↺). For each algorithm we give the number of distinct states

Table 2. Leader election algorithm in an arbitrary network

		A(287922, 1221437, 1)↻			B(287922, 1222805, 1)↻			C (47887, 134916, 0) ∅		
	Cou99	365	365	365	365	365	365	47887	134916	115
×	Cou99 shy-	365	1356	1358	365	1356	1358	47887	134916	226
×	Cou99 shy	365	1356	1358	365	1356	1358	47887	134916	226
	GV04	365	365	365	365	365	365	47887	134916	115
	CVWY90	17693	91145	902	448	789	787	47887	269831	115
	SE05	17693	90803	564	448	449	449	47887	269831	115
	Tau03	17702	187964	911	448	1876	787			
×	Tau03 opt	365	365	366	365	365	366	47887	134916	115
		D(289812, 1232783, 1)∅			E (145400, 413351, 0) ↻			F(289812, 1225799, 1)∅		
	Cou99	289812	1232783	145172	365	365	365	289812	1225799	145172
×	Cou99 shy-	289812	1232783	145666	365	706	708	289812	1225799	145666
×	Cou99 shy	289812	1232783	145304	365	706	708	289812	1225799	145304
	GV04	289812	1232783	145172	365	365	365	289812	1225799	145172
	CVWY90	289812	1642497	1145	365	703	704	289812	1635513	1145
	SE05	289812	1642497	1145	365	365	366	289812	1635513	1145
	Tau03	289812	2875280	1145				289812	2861312	1145
×	Tau03 opt	289812	1642497	1145	365	365	366	289812	1635513	1145
		G (241808, 687630, 1) ↻			H(728132, 2080615, 4)↻			I(728132, 2076619, 4)∅		
	Cou99	557	557	557	145847	413799	145172	728132	2076619	145172
×	Cou99 shy-	557	1087	1089	145847	414229	145303	728132	2076619	145307
×	Cou99 shy	557	1087	1089	145847	414229	145257	728132	2076619	145257
	GV04	557	557	557	145847	413799	145172	728132	2076619	145172
	CVWY90	557	895	896	178543	511930	1388	728132	2489217	1172
	SE05	557	557	558	178543	504468	1145	728132	2489217	1172
	Tau03	566	1249	905	178551	1604336	1454	728132	6631906	1454
×	Tau03 opt	557	557	558	145847	827149	1454	728132	4555287	1454

visited, traversed transitions, and the maximal size of the stack. No measures have been done for Tau03 on TGBAs without acceptance condition.

The complete reachability graph (i.e., without partial order reduction—the conclusion for the reduced graphs are similar, only with smaller figures) of the model has been generated from its Promela specification using Spin [13]. Then the corresponding TGBA has been introduced in Spot and the formulæ translated into TGBAs using also Spot. Though this is not generally the case, on this example the sizes of the degeneralized product and of the generalized one are identical. This is why Cou99 and GV04 perform equally well. The original implementation of Cou99 [3] would have used far less stack, but visited twice as many transitions (for instance on formula F the results for the implementation of Cou99 without rem are $\langle 289812, 2451598, 1145\rangle$).

These runs confirm the conclusions of Schwoon and Esparza [17]. SE05 always performs better than CVWY90 (formulæ A, B, E, G and H); and SCC-based algorithms Cou99 and GV04 perform better than NDFS ones (formulæ A and H).

To conclude our experimentation and focus on multiple acceptance conditions, we present complementary measures for a simple client-server example where c clients communicate with s servers via a duplex channel. Any client can send a request, then

Table 3. Client-server algorithm

		3 cl., 1 serv.			∅	3 cl., 1 serv., fair			∅
	Cou99	a	783	2371	511	b	783	2371	511
×	Cou99 shy-	a	783	2371	710	b	783	2371	710
×	Cou99 shy	a	783	2371	519	b	783	2371	519
	GV04	a	783	2371	511	b’	2005	6627	550
	CVWY90	a	783	2897	237	b’	2005	7771	251
	SE05	a	783	2897	237	b’	2005	7771	251
	Tau03	a	783	5268	238	b	783	10143	264
×	Tau03 opt	a	783	2897	237	b	783	8200	264
		3 cl., 3 serv.			↺	3 cl., 3 serv., fair			∅
	Cou99	c	631	839	159	d	21394	85387	11465
×	Cou99 shy-	c	631	1153	487	d	21394	85387	17133
×	Cou99 shy	c	1170	1914	401	d	21394	85387	11469
	GV04	c	631	839	159	d’	77979	339876	11521
	CVWY90	c	631	1513	159	d’	77979	410877	5632
	SE05	c	631	1499	159	d’	77979	410877	5632
	Tau03	c	899	3373	191	d	21394	415551	5099
×	Tau03 opt	c	631	1499	159	d	21394	331587	5060

sizes of products			
ref.	# st.	# tr.	# cond.
a	783	2371	1
b	783	2371	5
b’	2005	6627	1
c	21394	85387	1
d	21394	85387	7
d’	77979	339876	1

some server will answer that client. The property we check is that if the first client sends a request it will get an answer. This property is only satisfied for 1 client and is otherwise false unless weak fairness is assumed. Table 3 shows the measures. One can indeed see that the property is not satisfied in the case of 3 clients without fairness. The interesting point is that the additional acceptance conditions used for fairness constraints comes at no cost for Cou99 while the cost is high for other algorithms. This is obvious on the cases with 1 client (and can be generalized), however we cannot directly compare the product sizes for 3 clients as the fair case is empty while the unfair case is not.

4 Heuristics and Optimizations

4.1 Heuristics for SCC-Based Algorithms

The two shy variants of Cou99 measured in these tables use the fact that line c15 in Fig. 2 does not enforce any order on the successors. Cou99 will simply use the physical order of the successors in memory, so the *succ* member of *todo* items can be efficiently represented as an iterator. The Cou99 shy- variant orders successors to visit those that are already in H first before visiting new states. Doing so sounds natural because it favors merges of SCCs upon pushs. Cou99 shy works similarly, but it considers the successors of the whole top SCC instead of selecting a successor only among the successors of the state at the top of *todo* (in practice *todo* is merged like SCC).

Because Cou99 shy- and Cou99 shy have to reorder the successors before executing line c15, the *succ* field of *todo* entries cannot be represented as an iterator. To be fair our measures of the stack size of these two variants also account for the number of states of each *succ* field. Also, while Cou99 makes it possible to compute successors of a state one by one on-the-fly, this is not possible for shy variants who need *all* suc-

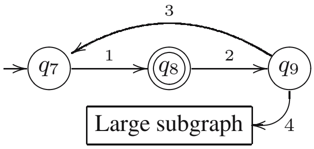

Fig. 4. Problematic case for SE05

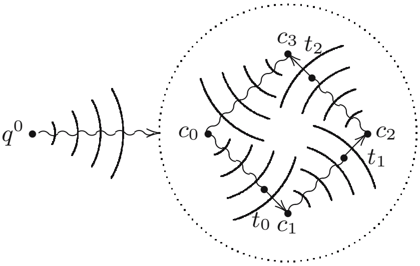

Fig. 5. Computing an accepting run for a TGBA

cessors to reorder them. This difference is apparent in the number of transitions visited: shy variants compute more transitions than plain Cou99.

These heuristics have a controversial effect on performance. Often, they will indeed visit less states, but in counterpart they compute more transitions and require more stack space. On non-empty automata, it is possible to find cases (e.g., bottom left of Table 3) where the variants visit more states. One issue with measuring on-the-fly emptiness checks is that they exit as soon as they can: a more complex algorithm may exit before an efficient one if it luckily picks successors in the right order. (Apart from these two shy variants, all the other algorithms implemented here visit states in the same DFS order; this ensures equitable measurements.) This confirms observations of Geldenhuys and Valmari [8] who tested other heuristics, none of which appeared better either.

4.2 A New Nested DFS Algorithm

Fig. 4 illustrates a case where SE05 could be improved. Arcs are labeled by their depth-first order. SE05 is defined on Büchi automata with accepting states. In its first DFS, if either q_9 or q_7 are accepting, then SE05 can report a violation. If q_8 is accepting, the accepting cycle (q_8, q_9, q_7, q_8) cannot be detected by the first DFS: it will only be found by the second DFS performed *after* the large subgraph have been explored.

The first DFS could detect an accepting cycle when visiting the third arc if it knew whether an accepting state exists between q_7 and q_9. We propose to associate each state q in the DFS stack with the number $W[q]$ of accepting states in the DFS path from q^0 to q. Therefore checking the existence of an accepting state between q_7 and q_9, amounts to testing whether $W[q_9] - W[q_7] > 0$.

This technique can be generalized to multiple acceptance conditions using a vector of counters. We implemented it in Tau03 opt. Its effect can be observed on TGBAs with a single acceptance condition, where SE05 and Tau03 opt differ only on this last optimization. For instance see formulæ A and B in Table 2.

Fig. 6 presents Tau03 opt. This new algorithm uses the technique of Tau03 to handle multiple acceptance conditions, but simplifies its logic and also implements all the optimizations introduced by SE05.

On Table 1 the reason why Tau03 opt outperforms GV04 in terms of visited states is that the latter works on a degeneralized automaton (this is confirmed when comparing Cou99 with Tau03 opt); however the way Tau03 opt nests multiple DFSs to handle multiple acceptance conditions causes more transitions to be visited than GV04.

```
// Let $\langle \Sigma, \mathcal{Q}, \delta, q^0, \mathcal{F} \rangle$ be the
// input automaton to check.
$H$: map of $\mathcal{Q} \mapsto \langle color \in \{\text{cyan}, \text{blue}\},$
                          $acc \subseteq \mathcal{F} \rangle$
$W$: map of $\mathcal{Q} \mapsto$ map of $\mathcal{F} \mapsto \mathbb{N}$
$weight$: map of $\mathcal{F} \mapsto \mathbb{N}$

main():
  forall $f \in \mathcal{F}$, $weight[f] \leftarrow 0$
  return dfs_blue($q^0$)

propagate($s \in \mathcal{Q}$, $Acc \subseteq \mathcal{F}$, $t \in \mathcal{Q}$):
  $\langle tcol, tacc \rangle \leftarrow H[t]$
  if $tcol = \text{cyan} \wedge \mathcal{F} = (H[s].acc \cup Acc \cup$
      $tacc \cup \{f \in \mathcal{F} \mid weight[f] > W[t][f]\})$
    return $\bot$
  else if $Acc \not\subseteq tacc$
    $H[t].acc \leftarrow tacc \cup Acc$
    if dfs_red($t$, $Acc$) $= \bot$
      return $\bot$
  return $\top$

dfs_blue($s \in \mathcal{Q}$):
  $H[s] \leftarrow \langle \text{cyan}, \emptyset \rangle$
  $W[s] \leftarrow weight$
  forall $\langle l, a, t \rangle$ such that $\langle s, l, a, t \rangle \in \delta$
    if $t \notin H$
      forall $f \in a$
        $weight[f] \leftarrow weight[f] + 1$
      if dfs_blue($t$) $= \bot$
        return $\bot$
      forall $f \in a$
        $weight[f] \leftarrow weight[f] - 1$
    if propagate($s$, $H[s].acc \cup a$, $t$) $= \bot$
      return $\bot$
  $H[s].color \leftarrow$ blue
  delete $W[s]$
  return $\top$

dfs_red($s \in \mathcal{Q}$, $Acc \subseteq \mathcal{F}$):
  forall $\langle l, a, t \rangle$ such that $\langle s, l, a, t \rangle \in \delta$
    if $t \in H \wedge$ propagate($s$, $Acc$, $t$) $= \bot$
      return $\bot$
  return $\top$
```

Fig. 6. A variation on the emptiness-check algorithm of Tauriainen [19]

5 Computing Accepting Runs for Generalized Automata

When a product space is found *not* to be empty, it means the system does not verify the formula it is checked against. An important step is to provide the user with a counterexample, showing an actual faulty execution of the system. Such a counterexample is an accepting run of the product automata. It can often be produced as a side-effect of the emptiness check, or afterwards by reusing some data of the check.

In emptiness-check algorithms that work on degeneralized automata, exhibiting an accepting run if one exists is straightforward. In NDFS-based algorithms (CVWY90, SE05) that run is the contents of the stack. For GV04, Geldenhuys and Valmari [9] showed how to use an extra integer per stack state to produce an accepting run.

In this section we present two techniques to extract accepting runs from the data structures of the algorithms that work on generalized automata: Cou99, Tau03, and their variants. Both techniques try to compute a short accepting cycle using successive breadth-first searches (BFSs) and then construct the shortest prefix leading to this cycle.

Accepting runs with Cou99. When Cou99 returns $\bot$ it means an accepting SCC is reachable from q^0. Fig. 5 shows this SCC as a dotted circle. A state s can easily be told to belong to this SCC by checking whether $H[s] \geqslant SCC.\text{top}().root$.

Because the SCC is accepting, from any of its states there exists a circuit labeled by all acceptance conditions. This circuit may cross the same transitions several times.

Table 4. Comparison of algorithms for computing accepting cycles

	Random formulæ				Human-generated formulæ			
Algorithm	formula's cond.		fair		formula's cond.		fair	
$d = 0.001$	*2328 (1318)*		*2188*		*2308 (2127)*		*1951*	
Cou99	1.9	2.0	17.1	12.7	1.3	1.3	12.3	9.0
Cou99 shy-	1.5	1.5	15.7	11.7	1.0	1.0	11.6	8.4
Cou99 shy	1.5	1.5	15.0	11.0	1.0	1.0	11.0	7.9
Tau03	10.9	9.9	237.6	64.7	8.7	9.5	205.9	53.9
Tau03 opt	2.8	6.8	64.5	18.5	2.2	7.4	53.4	16.4
$d = 0.002$	*2716 (1488)*		*2695*		*2569 (2304)*		*2548*	
Cou99	1.3	1.5	15.2	10.6	0.9	1.0	9.7	6.6
Cou99 shy-	0.9	0.9	14.0	9.4	0.6	0.7	8.8	5.9
Cou99 shy	0.9	0.9	13.3	8.6	0.6	0.6	8.5	5.6
Tau03	10.8	8.5	225.1	61.3	7.7	7.1	153.5	40.9
Tau03 opt	2.6	4.8	61.9	17.8	1.8	4.5	43.6	13.9
$d = 0.01$	*2978 (1569)*		*2979*		*2766 (2441)*		*2765*	
Cou99	0.8	1.0	12.5	7.4	0.5	0.6	7.1	4.2
Cou99 shy-	0.4	0.4	9.8	4.9	0.2	0.2	5.5	2.9
Cou99 shy	0.4	0.4	9.2	4.1	0.2	0.2	5.1	2.5
Tau03	18.1	16.9	210.0	58.1	12.6	11.2	139.4	35.5
Tau03 opt	1.7	3.5	43.0	12.4	1.1	2.6	25.6	7.8

Therefore, it is easier to construct an accepting cycle as a series of independent parts that can each visit a transition at most once, and that each brings new acceptance conditions.

The algorithm thus works as follows. Let $\mathcal{F}_0$ be the set of all acceptance conditions. From a state c_0 of the SCC, start a BFS (restricted to the SCC) to construct a path to the closest transition t_0 that has some acceptance conditions F_0 so that $F_0 \cap \mathcal{F}_0 \neq \emptyset$. Let $\mathcal{F}_1 = \mathcal{F}_0 \setminus F_0$ be the set of remaining acceptance conditions. Repeat the BFS from c_1 (the output of t_0) until a transition t_1 is found with acceptance conditions F_1 that intersect $\mathcal{F}_1$. Iterate until $\mathcal{F}_n = \emptyset$. Finally use a last BFS to compute the shortest path from c_n back to c_0, closing the cycle. This algorithm was presented by Latvala and Heljanko [15] using the root of the SCC as c_0. However the choice of c_0 can be arbitrary because we are in a SCC. Since we know that the transition that caused Cou99 to exit (the one corresponding to the last execution of line c15) is necessarily part of the acceptance cycle, it seems wiser to use either its source or its destination as c_0.

As far as the prefix is concerned, a list of states from q^0 to c_0 can be easily constructed while unwinding the *todo* stack. However this prefix may not be the shortest possible prefix leading to the accepting cycle, so a similar idea would be to use a BFS to construct the shortest path between q^0 and any state of the cycle, this path can be constrained to visit the SCCs in increasing order to limit the scope of the BFS.

Accepting runs with Tau03. Computing accepting runs for generalized NDFS algorithms such as Tau03 or Tau03 opt is more embarrassing, because the resulting data do not provide structural information as useful as a SCC that would restrict our search.We know that the last s for which line t34 was executed belongs to an accepting cycle. From this state $c_0 = s$ we first perform a nested DFS to collect a set of transitions $\mathcal{T}$ that (1) are each on a cycle back to c_0, and (2) will, together, cover $\mathcal{F}$.

This collection of cycles could be used to construct an accepting cycle, but since we are trying to create short runs we decided to connect these collected transitions directly. Therefore we perform a BFS to compute the shortest path from c_0 to a transition t_0 of $\mathcal{T}$, and from there another BFS to find the shortest path to another t_1 of $\mathcal{T}$, etc. Closing the cycle and computing the prefix can be done like for SCC-based algorithms.

Table 4 uses the layout of Table 1. For each setup, the two values are the number of states visited to construct the cycle part of the accepting run, and the size of the search space for this cycle. They are expressed as a percentage of the number of states of the input TGBA. For Cou99 the cycle's search space is the top SCC, and for Tau03 the search space contains all states in H. (A state is counted as many times as it is visited.)

As the table shows, the absence of structural information in Tau03 makes the computation more costly, since the search space is larger. For Cou99, the search is contained in a small subgraph (the top SCC), which justifies the use of BFSs. The "fair" columns show that with more acceptance conditions in the system the algorithms need to traverse the search space more times. Surprisingly, the size of the search space for Cou99 shy and Cou99 shy- is smaller than that of Cou99; this is counter-intuitive because our heuristics aim at favoring merges of SCCs.

During our experiments we observed that the size of accepting runs produced by such BFS-based algorithms were significantly smaller than those obtained directly from the stack of NDFS algorithms. A deeper study of existing algorithms, weighting minimization against computational complexity still has to be done (Gastin et al. [7] provide some initial clues).

6 Conclusion

In this paper we have stressed the importance of dealing with *generalized* Büchi automata in emptiness-check algorithms. Our experiments on existing algorithms showed that SCC-based ones clearly outrank NDFSs; this completes the results of Schwoon and Esparza [17], who studied emptiness checks of standard Büchi automata.

Although we have not implemented it, the generalized algorithms presented here can be used in conjunction with the bit-state hashing technique [13, p. 206] if done carefully. The bit-state hashing should not be applied to states that belong to the first-level DFS: those states need to be perfectly hashed. The application to Tau03 is discussed by Tauriainen [20]. In SCC-based algorithms the restriction extends to all states that belong to SCCs in the SCC stack. In other words, bit-state hashing can only be applied to states from removed MSCCs; this limits its usefulness.

To give NDFS-based algorithms a chance to compete with SCC-based ones, we introduced (1) a new optimization to detect some accepting cycles earlier, and (2) a new algorithm (Tau03 opt) that mixes all the optimizations of SE05 with the multiple acceptance condition capability of Tau03. Although Tau03 opt surpasses other NDFS algorithms, our experiments still show that SCC-based algorithms perform better.

To complete our TGBA verification framework, we finally introduced algorithms to extract accepting runs. Here again, our results are in favor of Cou99.

All the algorithms presented and measured here are implemented in our model-checking library, Spot [4], available at `http://spot.lip6.fr`. The distribution of Spot includes the scripts we used for our experiments. They can be adjusted to different configurations, and can output more statistics than we could present in these pages. For instance they also verify the reduced state spaces generated from the examples using Spin's partial-order reduction [13, p. 192].

References

[1] I. Černá and R. Pelánek. Relating hierarchy of temporal properties to model checking. In *Proc. of MFCS'03*, volume 2747 of *LNCS*, pages 318–327. Springer-Verlag, Aug. 2003.

[2] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithm for the verification of temporal properties. In *Proc. of CAV'90*, volume 531 of *LNCS*, pages 233–242. Springer-Verlag, 1991.

[3] J.-M. Couvreur. On-the-fly verification of temporal logic. In *Proc. of FM'99*, volume 1708 of *LNCS*, pages 253–271. Springer-Verlag, Sept. 1999.

[4] A. Duret-Lutz and D. Poitrenaud. Spot: an extensible model checking library using transition-based generalized Büchi automata. In *Proc. of MASCOTS'04*, pages 76–83. IEEE Computer Society, Oct. 2004.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proc. of FMSP'98*, pages 7–15. ACM, Mar. 1998.

[6] K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In *Proc. of Concur'00*, volume 1877 of *LNCS*, pages 153–167. Springer-Verlag, 2000.

[7] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexamples in SPIN. In *Proc. of SPIN'04*, volume 2989 of *LNCS*, pages 92–108. Springer-Verlag, 2004.

[8] J. Geldenhuys and A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 205–219. Springer-Verlag, 2004.

[9] J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoretical Computer Science*, 2005. To appear: conference paper selected for journal publication.

[10] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulæ to Büchi automata. In *Proc. of FORTE'02*, volume 2529 of *LNCS*, pages 308–326. Springer-Verlag, Nov. 2002.

[11] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Proc. of PSTV'93*, volume C-16 of *IFIP Transactions*, pages 109–124. North-Holland, May 1993.

[12] M. Hammer, A. Knapp, and S. Merz. Truly on-the-fly LTL model checking. In *Proc. of TACAS'05*, LNCS. Springer-Verlag, Apr. 2005.

[13] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. ISBN 0-321-22862-6.

[14] G. J. Holzmann, D. A. Peled, and M. Yannakakis. On nested depth first search. In *Proc. of SPIN'96*, volume 32 of *DIMACS*. AMS, May 1996.

[15] T. Latvala and K. Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43 (1–4):1–19, 2000.

[16] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. of POPL'85*, pages 97–107. ACM, 1985.

[17] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *Proc. of TACAS'05*, LNCS. Springer-Verlag, Apr. 2005. To appear.

[18] F. Somenzi and R. Bloem. Efficient Büchi automata for LTL formulæ. In *Proc. of CAV'00*, volume 1855 of *LNCS*, pages 247–263. Springer-Verlag, 2000.

[19] H. Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Research Report A83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, Dec. 2003.

[20] H. Tauriainen. Nested emptiness search for generalized Büchi automata. In *Proc. of ACSD'04*, pages 165–174. IEEE Computer Society, June 2004.

[21] H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulæ into Büchi automata. In *Proc. of CS&P'99*, pages 251–262, Sept. 1999.

[22] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of Banff'94*, volume 1043 of *LNCS*, pages 238–266. Springer-Verlag, 1996.