Coping With Strong Fairness – On-the-fly Emptiness Checking for Streett Automata

Timo Latvala <Timo.Latvala@hut.fi>
Keijo Heljanko*<Keijo.Heljanko@hut.fi>
Laboratory for Theoretical Computer Science
Helsinki University of Technology
P.O. Box 5400
FIN-02015 HUT
Finland

Abstract

The model checking approach to verification has proven to be very successful. Stating the required properties in temporal logic allows the process of verification to be automated. When propositional linear temporal logic (LTL) is used as the specification language Büchi automata usually are the theoretical constructions of choice. Büchi automata, however, cannot cope with strong fairness efficiently. Streett automata can easily model strong fairness constraints but the known model checking algorithms are more complex than the algorithms for Büchi automata. In order to circumvent this problem we present a method which takes advantage of the simpler model checking complexity of Büchi automata when possible, and uses Streett automata when faced with strong fairness constraints. The method performs the model checking in an on-the-fly fashion and also includes counterexample generation. We present memory efficient algorithms for both the emptiness checking of Streett automata and counterexample generation.

*Currently visiting Technische Universität München, Fakultät für Informatik

1 Introduction

The systems being developed today are more complex than ever. Distribution and concurrency are effective tools for solving many of the problems developers are presented with. Unfortunately they also introduce new problems in the form errors which can be extremely hard to find.

Formal methods has given us new tools to find these bugs. However verification, i.e. checking that a system conforms to its specification, is a difficult task. Simple testing has endured as a validation method although it has proven to be a both time consuming and not always a successful measure. The model checking approach has emerged as a viable alternative. Model checking uses the equivalence between automata on infinite words and temporal logic to verify specifications. Given a specification in a temporal logic and a description of the systems executions, model checking automates the process of verification.

The applicability of model checking is however seriously limited by the state-space explosion problem. Due to this verifying complex systems is in many cases an intractable problem. Several remedies have, however, been suggested for the problem. One such remedy is performing the model checking on-the-fly. When performing model checking on-the-fly the state-space is constructed simultaneously while the model checking is done. This means that errors might be found before the construction of the complete state space.

Büchi automata are frequently used when the specification language used is linear-time propositional temporal logic (LTL). Several schemes have been presented on how to perform on-the-fly verification using Büchi automata [CVWY92, Kur94]. However Büchi automata have problems with efficiently representing systems which contain strong fairness constraints. Since efficient modeling of systems requires strong fairness [Fra86], coping with it in an efficient manner would be desirable.

A class of automata that can handle strong fairness efficiently are the Streett automata, i.e. the complemented pairs automata, see e.g. [Tho97]. In this paper we present a method of how to perform verification in an on-the-fly fashion using a combination of Büchi and Streett automata. The method provides efficient handling of both weak and strong fairness constraints by avoiding the greater complexity of model checking for Streett automata when possible. We also present a memory efficient algorithm for performing the emptiness check on the Streett automata. In recognition of the importance of providing a counterexample we also present a new and efficient algorithm which finds short counterexamples when the formula does not hold.

2 Automata on Infinite Words

The theory of automata on infinite words provides the theoretical foundation for model checking.

Definition [Tho97] A *finite ω-automaton* is a tuple $\mathcal{A}$=$(S, \Sigma, s_0, \Delta, \Omega)$ where, S is a finite nonempty set of states, Σ the input alphabet, $s_0 \in S$ the initial state, Ω the acceptance component and $\Delta \subset S \times \Sigma \times S$ is the transition relation. A run of $\mathcal{A}$ on a given input ω-word $\alpha = \alpha(0)\alpha(1)...$ with $\alpha(i) \in \Sigma$ is a sequence $\rho = \rho(0)\rho(1)... \in S^\omega$ such that $\rho(0) = s_0$ and $(\rho(i),\ \alpha(i),\ \rho(i+1)) \in \Delta$ for $i \geq 0$.

By denoting the quantifier "there exist infinitely many" by $\exists^\omega$ we can define the set

$$In(\rho) = \{s \in S | \exists^\omega i\ \rho(i) = s\}.$$

Consequently $In(\rho)$ is the set of states that occur infinitely often in the run ρ. A *Büchi* automaton, see e.g. [Tho97], is then obtained from the ω-automaton if we define for the acceptance component the following condition, the Büchi condition: $In(\rho) \cap F \neq \emptyset$ for a set $F \subseteq S$ of accepting states. This means that for an accepting run some state from the set of accepting states must be visited infinitely often.

A *generalized Büchi* automaton [Gri88] differs from an ordinary Büchi automaton in that its acceptance component is a finite set of acceptance sets $\mathcal{F} = \{F_0, F_1, \ldots, F_{n-1}\} \subseteq 2^S$. A run $\rho = s_0 s_1 \ldots$ of a generalized Büchi automaton is accepting if and only if for all accepting sets $F_j \in \mathcal{F}$ there exists some state $s_{k_j} \in F_j$ such that it occurs infinitely many times in the run. Compactly expressed, for an accepting run $In(\rho) \cap F_i \neq \emptyset$ for each $F_i \in \mathcal{F}$.

Fairness was already mentioned as a useful tool in modeling systems. In [Fra86] fairness is defined using the notion of *enabledness* of relevant events. *Weak fairness* requires that an event will not be indefinitely postponed if it is continuously enabled. A weakly fair Scheduler with a waiting queue, is guaranteed to eventually schedule an event once the event has entered the queue. Other examples where weak fairness is appropriate are systems with busy waiting and many resource-allocation processes. Weak fairness does not cover all situations encountered in modeling. A usual assumption made for communication protocols is that if a message is sent repeatedly it will eventually be successfully received. This assumption could be violated although the communication process would be weakly fair. *Strong fairness* requires that if an event if infinitely often enabled it must occur infinitely often. With strong fairness the assumption made for communications protocols is satisfied. These definitions of fairness can be adapted to suit most formalisms. Büchi automata, which are commonly used for on-the-fly model checking, can represent weak fairness efficiently. The question arises: how can strong fairness be efficiently represented?

A *Street* automaton, see e.g. [Tho97], is obtained with the Streett, i.e. the complemented pairs, acceptance condition if we define for the acceptance component: $\bigwedge_{i=1}^{k}(In(\rho) \cap L_i = \emptyset \vee In(\rho) \cap U_i \neq \emptyset)$ for a sequence Ω of pairs $(L_1, U_1), \ldots, (L_k, U_k)$, where L_i and U_i are subsets of S.

Clearly the acceptance condition represents strong fairness conditions as defined in [Fra86] and can be read as "for each i, if some state in L_i is visited infinitely often, then some state in U_i is visited infinitely often". Obviously Streett automata are the answer to our previous question.

The definitions above are automata where the labels are on the arcs. The algorithms presented later, however, use automata where the states are labeled. It is easy to see that arc labeled automata can easily be transformed into state labeled automata and vice versa.

The set of infinite strings an automaton accepts is denoted by $\mathcal{L}(\mathcal{A})$, and is called the language of $\mathcal{A}$. Testing if the language is empty, denoted by $\mathcal{L}(\mathcal{A}) = \emptyset$, is referred to as performing an emptiness check.

3 Model Checking

Because it is possible to interpret the behavior of a finite state system as a set of computations, it is possible to use the equivalence between linear temporal logic and automata on infinite words. A computation is a function $\pi : \mathbb{N} \to 2^{Prop}$, where $Prop$ is a given finite set of atomic propositions. The function assigns truth-values to the propositions at each time instant $i \in \mathbb{N}$. The atomic propositions describe the internal state of the system. *Linear-time propositional temporal logic* is also interpreted over computations. Hence, LTL can be used for specifying properties of reactive, non-terminating systems.

Computations as can also be viewed as infinite words. In [SPH84] it was shown that there is an automaton on infinite words that accepts exactly the computations that satisfies an LTL formula. Later in [VW94] an explicit construction was given for how to convert an LTL formula φ into a Büchi automaton $\mathcal{A}_\varphi$, which accepts exactly the same computations. A refined algorithm was presented in [GPVW95] which performed the translation to a generalized Büchi automaton on-the-fly. That the automaton accepts the same computations as the formula implies that satisfiability testing is equivalent to doing an emptiness check of an automaton. It also gives us a method to do model checking using automata-theoretic constructions.

The steps performed to verify that a system has a property given by a LTL formula φ are the following [CVWY92, Kur94].

1. Generate the reachability graph of the system and interpret it as a Büchi automaton $\mathcal{S}$.
2. Construct the automaton $\mathcal{A}_{\neg\varphi}$ corresponding to the negation of the property φ.
3. Form the product automaton $\mathcal{B} = \mathcal{A}_{\neg\varphi} \times \mathcal{S}$.
4. Check if $\mathcal{L}(\mathcal{B}) = \emptyset$.

If $\mathcal{L}(\mathcal{B}) = \emptyset$ the system satisfies the specification. The combination of some of these steps is referred to as "on-the-fly". There are several algorithms available on how to do this on-the-fly with Büchi automata [CVWY92, Kur94].

A limitation of the on-the-fly method described in [CVWY92] is that it can only deal with normal (non-generalized) Büchi automata. To handle generalized Büchi automata, the algorithm needs to translate these into Büchi automata, see e.g. [GPVW95]. This can also be done on-the-fly, but the number of states of the resulting product automaton can be the number of states of the original product automaton times the number of generalized Büchi acceptance sets.

As previously mentioned Büchi automata cannot handle strong fairness efficiently, as there is no polynomial translation form Streett to Büchi automata, see e.g. [Saf89]. Weak fairness, however, is manageable with generalized Büchi automata. By combining the best of both worlds it is possible to deal with both strong and weak fairness and verify claims given in LTL in an efficient manner. We propose the following procedure:

1. Construct the generalized Büchi automaton $\mathcal{A}_{\neg\varphi}$.

2. The reachability graph of the system is transformed into an automaton with both Streett (for strong fairness) and generalized Büchi (for weak fairness) acceptance conditions, which are defined in the formal system description.

3. The product automaton of the above two is created on-the-fly, with states obtaining acceptance conditions from both the formula- and the reachability graph-automaton. Simultaneously Tarjan's algorithm is used to calculate the next maximal strongly connected component (MSCC) of the product automaton.

4. When a MSCC of the product automaton has been calculated we check for generalized Büchi acceptance. If the component does not contain a state from each Büchi acceptance set, i.e. it does not contain a weakly fair counterexample and hence is not accepted, we return to step 3.

5. If a component is accepted as weakly fair and it does not contain strong fairness constraints we can directly generate a counterexample at step 7 using only generalized Büchi sets interpreted as Streett acceptance sets U_i and with each L_i set initialized to the universal set.

6. The MSCC is checked by the Streett emptiness check, with the generalized Büchi acceptance sets interpreted as in step 5. If no weakly and strongly fair execution is found, we continue from step 3.

7. A counterexample is generated using the subset of vertices of the MSCC, which the emptiness checking algorithm supplies.

Note that steps 4 and 5 are not needed for correctness, they are only an optimization to do the Streett emptiness check only when it is needed. By performing the verification in this manner it is possible to find errors without computing all MSCCs, which might result in faster running times. Also the fact that only components which are weakly fair and contain strong fairness requirements are checked for strong fairness, potentially results in less work compared to a naive implementation.

4 Emptiness Checking of Streett Automata

The emptiness algorithm is given a MSCC, S, of the product Streett automaton. The product Streett automaton can be seen as directed graph $G = (V, E)$, where the number of vertices $|V| = n$ and the number of edges $|E| = m$. The Streett pairs (L_i, U_i), $L_i, U_i \subseteq V$ with $1 \leq i \leq k$ are given and $bits(S)$ is defined as $\Sigma_{i=1}^{k} |L_i| + |U_i|$ for $S \subseteq V$. Performing an emptiness check on a Streett automaton is then to see whether G contains a cycle such that: if the cycle contains a vertex from L_i then it also contains a vertex from U_i, for all $i \in \{1, \ldots, k\}$.

4.1 Emptiness Checking Algorithm

The main idea of the emptiness checking algorithm goes back to at least Emerson and Lei [EL87] and it was also independently developed in [LP85]. The algorithm is given a maximal strongly connected component (MSCC) of the product automaton calculated by a modified version of Tarjan's algorithm [Tar72]. The algorithm begins dynamically modifying the graph by deletion of *bad* vertices from the MSCC. A vertex is bad if it belongs to some L_i set, but the MSCC it belongs to does not contain a vertex from the corresponding U_i set. All other vertices are good. A MSCC containing only good vertices is said to be a good component. After the deletion of bad vertices the MSCC of the modified graph are now recalculated and again checked for bad vertices. The algorithm terminates when it has found a non-trivial good component or it concludes that no such component exists.

The algorithm checks if a MSCC is trivial by first checking the size of the component. If it is a single vertex the algorithm checks if it has an edge to itself. If the single vertex does not have an edge to itself the MSCC is trivial. If the single vertex has an edge to itself or the number of vertices in the component is two or more the MSCC is non-trivial.

The algorithm presented here is similar to [RT97]. The data structures used are simpler, but by modifying them and adding the lock-step search case of their algorithm, we would achieve the same running time.

4.2 Data Structures

The algorithm needs to keep track of bad vertices and into which MSCC different vertices belong to as the original MSCC may split into several MSCCs. It is also necessary to keep track of which fairness sets are present in the currently processed component. The notation in this section has been chosen in order to be consistent with [RT97].

We use three global sets L, U and $Badsets$ of size k which are implemented as a combination of a stack and a bitmap. They require a one-time initialization which takes time $O(k)$. This is done the first time the emptiness algorithm is called. With this implementation set membership can be tested in $O(1)$ time. Set union $A := A \cup B$, difference $A := A \setminus B$, and set clear $B := \emptyset$ can be done in $O(|B|)$ time.

The data structure $C(S)$ stores the component information and for each vertex the information concerning L_i and U_i membership. It is implemented with a doubly linked list which contains the vertices of the MSCC and their respective component numbers. From each node in the list there are pointers to set lists which specifies to which L_i and U_i sets the vertex belongs to. These sets are referred to as $L.setlist$ and $U.setlist$ respectively.

As the original component may split into several MSCCs during the run of the algorithm a queue Q is kept where the different components are stored. We define the following operations for the data structure $C(S)$:

Construct(S) initializes and returns the data structure $C(S)$.
Remove$(C(S),\, B)$ removes B from S and returns $C(S \backslash B)$ for $B \subseteq S \subseteq V$.
Bad$(C(S))$ returns $\bigcup_{1 \leq i \leq k} S \cap L_i | S \cap U_i = \emptyset$ for $S \subseteq V$.

Lemma 1. *The operation Construct*(S) *can be implemented with a running time of* $O(|S|)$.

Proof. The given vertex list S is traversed. For each vertex an entry in the doubly linked list is created.

Lemma 2. *The operation Remove*$(C(S), B)$ *can be implemented with a running time of* $O(|B|)$.

Proof. Traversing the given list of bad vertices, B, and removing each entry in $C(S)$ takes time $O(|B|)$.

Lemma 3. *The operation Bad*$(C(S))$ *can be implemented with a running time of* $O(|S| + bits(S))$.

Proof. Traverse the set lists of each vertex in $C(S)$. Whenever a vertex is member of an L_i set or a U_i set add the set number to L or U respectively. This takes time $O(|S| + bits(S))$. Form the set $Badsets = L \backslash U$ and reset L and U. This can be done in time $O(min(k, bits(S)) = O(bits(S))$. Add all vertices to a list of bad vertices for which $L.setlist \cap BadSets \neq \emptyset$, reset $Badsets$ and then return the list. This takes time $O(|S| + bits(S))$ giving a total running time of $O(|S| + bits(S))$.

We believe that the simple data structures we use for emptiness checking will use less memory than the more sophisticated data structures of [RT97], and the fact that we do not require the predecessor relation for the reachability graph will compensate for the worse worst-case running time of our emptiness checking algorithm.

Theorem 4. *The emptiness algorithm will find a good component if it exists.*

Proof. The main loop of the algorithm maintains the invariant that all vertices are either bad, trivial or still in the queue. The algorithm initially puts all vertices in the queue. A vertex can leave only by being removed by the Remove function in the second while loop or by deemed as trivial. All other vertices are always put back in the queue. Hence the invariant holds and the algorithm will find a good component if it exists.

Theorem 5. *The running time of the algorithm without the Counterexample algorithm is* $O((m + bits(V))\ min(n, k))$

Proof. The total cost of the calls to Tarjan's algorithm is $O(m\ \min(n, k))$ because before each call at least one vertex and one fairness set has been taken care of. The same factor $\min(n, k)$ bounds the number of calls to *Bad*, *Construct*, *Remove*. Hence they contribute $O((n + bits(V))\ \min(n, k)) = O((m + bits(V))\ \min(n, k))$ to the running time giving a total of $O((m + bits(V))\ \min(n, k))$.

Theorem 6. *The memory usage of the emptiness algorithm is bounded by* $O(m+n+bits(V)+k)$

```
proc Empty(S, k) ≡
  Queue Q_1, Q_2;
  List B;
  boolean change;
  InitSets(k);                                              Initialize sets L, U, Badsets
  C(S) := Construct(S);
  put(Q_1, C(S));
  while (Q_1 ≠ ∅) do
        C(S) := get(Q_1);
        change := false;
        while (B:=Bad(C(S)) ≠ ∅) do
              C(S) := Remove(C(S), B);
              change := true;
        od
        if (change AND C(S) ≠ ∅) then
                                      Tarjan(C(S), Q_2);            recalculate the MSCCs
                                      RemoveLargestMSCC(Q_2);
                                      while Q_2 ≠ ∅ do
                                            B := get(Q_2);
                                            C(S) := Remove(C(S), B);
                                            put(Q_1, Construct(B)));
                                      od
                                      put(Q_1, C(S));
                                 else                              Good component found!
                                      if (NotTrivial(C(S)))
                                        Counterexample(C(S)); Generate a counterexample
                                        return true;
                                      fi
          fi
        od
        return false;                                     No good component exists
  .
```

Figure 1: The emptiness checking algorithm

Proof. The memory for representing the vertices and the edge information accounts for the term $n+m$. The memory required for the $C(S)$ data structure with the Streett set information amounts to $O(n + bits(V))$. Finally the sets $Badsets$, L and U use $O(k)$ memory giving a total of $O(n + m + k + bits(V))$.

4.3 The Modified MSCC search

The MSCC are found by using a modified version of Tarjan's algorithm [Tar72]. The algorithm is non-recursive and is based on a similar algorithm found in [Hel97]. It performs a search restricted to a particular $C(S)$ component for greater efficiency. When a MSCC is found the states of that MSCC are stored in a list, which is put into a queue, Q_2.

4.4 The Counterexample Algorithm

Generating a counterexample to the given property is very important to ease the location of design errors. The counterexample algorithm given here produces a counterexample after the emptiness algorithm has passed it a good component. Finding a counterexample is non-trivial because the counterexample can be a cycle which contains several loops.

The algorithm searches in a breadth-first manner from the MSCC entry vertex, which we will henceforth refer to as the root, for a path back to the root. The path must of course satisfy the requirement that if there is a vertex $v_i \in L_l$ in the path, the path must also include a vertex $v_j \in U_l$, for all $1 \leq l \leq k$. As the breadth-first search spawns a path tree, one must choose which path to use. The algorithm freezes the path traversed to the current vertex when

- the vertex belongs to an unseen L_i set
- the vertex belongs to an unseen U_i set corresponding to a previously encountered L_i set.

The traversed path is then printed from memory, the breadth-first search state is reset using logs and then search for the root can proceed. While the resetting adds to the running time of the algorithm it allows one to minimize memory requirements because the algorithm only keeps at most one simple cycle of the path traversed in memory. The algorithm terminates if it reaches the root and the traversed path satisfies the requirements. To know when to terminate, the algorithm keeps track of the number of encountered L_i sets for which the corresponding U_i set has not been found using the variable $unseen_L$.

The function that determines whether to freeze the breadth-first search is called checkstate. It returns true if we are in a vertex $v \in L_i$, and no state belonging to L_i has been seen before. It also returns true if the vertex v belongs to an unseen U_i set for which a corresponding vertex $u \in L_i$ has already been seen. The printing of the path from memory is done by $lockpath(s)$. This function also marks all U_i sets in the locked path which have not been encountered before as seen U_i sets, and resets the breadth-first search state.

An interesting special case occurs if the component contains no vertex for which $v \in \bigcup_{1,\ldots,k} L_i$. In this case the search reduces to a simple breadth-first search for a path back to the root. This can be done in linear time and space. The path found is also optimal in the sense that it involves the minimum number of vertices.

Lemma 7. *The Counterexample algorithm finds the counterexample when given a good component with no vertex belonging to a L_i set, and its running time is $O(n+m)$.*

Proof. Because no vertex belongs to an L_i set the algorithm will not reset. Hence it will find a path to the root, which is the counterexample and as the algorithm proceeds in a breadth-first manner one can achieve the running time $O(n+m)$.

```
proc checkstate(s) ≡
   Set seen_L;
   Set seen_U;
   boolean lockpath = false;
   integer unseen_L;
   forall v ∈ s.L.setlist do
                  if (v ∉ seen_L) then
                                 seen_L := seen_L ∪ {v}
                                 lockpath := true;
                                 if (v ∉ seen_U) then
                                                unseen_L++;
                                 fi
                  fi
   od
   forall v ∈ s.U.setlist do
                  if (v ∉ seen_U AND v ∈ seen_L) then
                                           unseen_L−−;
                                           lockpath := true;
                  fi
   od
   return lockpath;
.
```

Figure 2: The checkstate algorithm which determines when the BFS path is frozen

Lemma 8. *The running time of checkstate(s) is $O(bits(s))$ for $s \in S$.*

Proof. The function traverses the set list of the state and can in $O(1)$ time check if a specific set has been taken care of. The time required for the traversal is $O(bits(s))$.

Lemma 9. *The running time of lockpath(s) is $O(|S| + bits(S))$*

Proof. The function must reset the log storing the path, and go through the set lists of the vertices in the path and mark all unseen L_i sets encountered as seen. This gives a running time of $O(|S| + bits(S))$.

Theorem 10. *The Counterexample algorithm for the second case always finds a counterexample when given a good component, and its running time is $O((m + bits(S))\ min(n, k))$.*

Proof. The algorithm stores the traversed path up to the reset. After the reset any state is again visitable (the states are always reachable as we are traversing a MSCC). The algorithm will always find a new $s_i \in L_l$, or a corresponding $s_j \in U_l$ after a reset because all states are reachable and visitable and a new reset will not be performed unless any of the above are found or it enters the root and can terminate. Hence the algorithm will always find an accepting path given a good component. The algorithm performs $\min(n, 2k)$ resets in the worst case. Consequently the algorithm may have to traverse the graph and perform a checkstate at most $\min(n, 2k)$ times. This gives a total running time of $O((n + m + bits(S))\ \min(n, k))$.

```
proc Counterexample(C(S)) ≡
   Queue Q;
   state s, root, t;
   put(Q, root := root(C(S)));
   PrintPathTo(root);
   visit(root);
   while (Q ≠ ∅) do
         s := get(Q);
         if (checkstate(s)) then                              Do we freeze the BFS?
                        lockpath(s);            print path and reset the BFS state
         fi
         forall  t ∈ succ_in_comp(s) do                          Check if we are done
                        if (t = root AND unseen_L_i = 0) then
                                                   return ;
                        fi
         od
         forall  t ∈ succ_in_comp(s) do   put the successors of the vertex in the queue
                        if (¬(visited(t)) then
                                    visit(t);
                                    put(Q, t);
                                    log_father(t);  store the path
                        fi
         od
   od
.
```

Figure 3: The counterexample algorithm.

The maximum length of the counterexample is $n \min(n, 2k)$. There is a variation of the algorithm which always goes trough all U_i sets and hence generates a counterexample which has a maximum length of $n \min(n, k)$ [KPR98]. However, we suspect that in practice it would perform worse than the previously presented lazy algorithm. Deciding whether there exists a counterexample of length n, where n is the number of nodes is in fact NP-complete [CGMZ94] (proof with a reduction from a Hamiltonian cycle problem).

There a few papers which have included counterexample generation from an accepting component of a Streett automaton. Kesten, Pnueli and and Raviv [KPR98], present a non-lazy algorithm compatible with BDDs, which in some cases may produce shorter counterexamples. However, as the algorithm always searches for all fairness sets it will in many cases produce longer counterexamples. Another paper which also presents a counterexample algorithm for Streett automata is [HSB94].

Theorem 11. *The memory usage of the counterexample algorithm is bounded by $O(n+m+k+bits(S))$.*

Proof. The functions *lockpath* and *checkstate* can use the same sets *Bad*, L and U for their bookkeeping as the emptiness algorithm. Consequently the algorithm does not need additional data structures to those already created by the emptiness algorithm, except for a breadth-first search log and father log, which only incurs a linear penalty in the number of states n.

5 Conclusions

Including both strong and weak fairness usually complicates model checking. Most methods require that the strong fairness constraints are given in the properties to be checked [CVWY92, Kur94]. Giving the constrains in the system description is much more efficient [KPR98]. We have presented a complete method of how to perform an on-the-fly verification in an efficient manner when both weak and strong fairness assumptions are present in the system description. The method allows for faster error detection in some cases compared to methods relying exclusively on the use of Streett automata. As counterexample generation is very important for debugging purposes, we also presented a new memory conservative algorithm for finding short counterexamples when the formula to be verified does not hold. To our knowledge the algorithm should be both faster and produce shorter counterexamples in most cases than previously existing algorithms.

We have implemented the emptiness checking algorithm and the counterexample algorithm in Java(tm) and tested it with randomly generated SCCs. A complete verision of the model checking algorithm will be implemented in a new analyzer for Many-Sorted Petri Nets, Maria (http://www.tcs.hut.fi/html/Maria.html).

6 Acknowledgments

This research was funded by the National Technology Agency (former Technology development center of Finland, Tekes), Nokia Research Center, Helsinki Telephone Corporation and the Finnish Rail Administration. The second author also gratefully acknowledges the financial support of Helsinki Graduate School on Computer Science and Engineering (HeCSE), the Academy of Finland (Project 8309), the Support Foundation of Helsinki University of Technology, and the Eemil Aaltonen Foundation.

The authors also wish to thank Tommi Junttila for helpful criticisms and comments.

References

[CGMZ94] E. Clarke, O. Grumberg, K. McMilllan and X. Zhao. Efficient Generation Counterexamples and Witnesses in Symbolic Model Checking. *Technical Report TR CMU-CS-94-204*, Carnegie Mellon University, School of Computer Science, Pittsburg, 1994.

[CVWY92] C .Courcoubetis, M. Vardi, P. Wolper and M. Yannakakis.Memory-Efficient Algorithms for the Verfication of Temporal Properties. *Formal Methods in System Design*, vol 1. pp. 275-288, 1992.

[EL87] E.A. Emerson and C-L. Lei. Modalities for Model Checking: Branching Time Logic Strikes Back. *Science of Computer Programming*, vol. 8, no. 3, pp 275-306, 1987.

[Fra86] N. Francez. *Fairness.* Springer Verlag, New York, 1986.

[GPVW95] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. *Proceedings of the 15th Workshop on Protocol Specification, Testing and Verification*, pp. 3-18. Chapman and Hall, Warsaw Poland, 1995.

[Gri88] P. Gribomont. Temporal Logic, in: *From Modal Logic to Deductive Databases* (A. Thayse Ed.), pp. 165-234, John Wiley & Sons, Chichester, 1988.

[Hel97] K. Heljanko. Model Checking the Branching Time Temporal Logic CTL. *Research Report A45*, Digital Systems Laboratory, Helsinki University of Technology, 1997.

[HSB94] R. Hojati, V. Singhal and R.K. Brayton. Edge-Strett/ Edge-Rabin Automata Environment for Formal Verification Using Language Containment. *Memorandum No. UCB/ERL M94/12,* Electronics Res. Lab., Cory Hall, University of California, Berkeley, 1994.

[KPR98] Y. Kesten, A. Pnueli and L. Raviv. Algorithmic Verification of Linear Temporal Properties. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP 1998)*, Lecture Notes in Computer Science, vol. 1443, pp. 1-16. Springer-Verlag, 1998.

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, Princeton, New Jersey, 1994.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pp 97-107, 1985.

[RT97] M. Rauch Henzinger, J. Telle. Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory(SWAT'96)*, pp. 10-20. 1997.

[Saf89] S. Safra. *Complexity of Automata on Infinite Objects*, PhD Thesis, The Weizmann Institute of Science, 1989.

[SPH84] R. Sherman, A. Pnueli and D. Harel. Is the Interesting Part of Process Logic Uninteresting: a Translation From PL to PDL. *SIAM Journal on Computing*, vol. 13, no. 4, pp. 825-839, 1984.

[Tar72] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, vol. 1, no. 2, pp 146-160, 1972.

[Tho97] W. Thomas. Languages, Automata and Logic, in: *Handbook of Formal Languages* (G. Rozenberg, A. Salomaa, Eds.). Vol III, pp. 385-455, Springer-Verlag, New York, 1997.

[VW94] M.Y. Vardi, P. Wolper. Reasoning About Infinite Computations. *Information and Computation*, vol. 115, no. 1, pp. 1-37, 1994.