Efficient Model Checking of Safety Properties

Timo Latvala [*]

Laboratory for Theoretical Computer Science
Helsinki University of Technology
P.O. Box 9205
FIN-02015 HUT
Finland
Timo.Latvala@hut.fi

Abstract. We consider the problems of identifying LTL safety properties and translating them to finite automata. We present an algorithm for constructing finite automata recognising informative prefixes of LTL formulas based on [1]. The implementation also includes a procedure for deciding if a formula is pathologic. Experimental results indicate that the translation is competitive when compared to model checking with tools translating full LTL to Büchi automata.

1 Introduction

Informally, safety properties are properties of systems where every violation of a property occurs after a finite execution of the system. Safety properties are relevant in many areas of formal methods. Testing methods based on executing a finite input and observing the output can only detect safety property violations. Monitoring executions of programs is also an area where safety properties are relevant as monitoring also only can detect failures of safety properties. Naturally, formal specifications are also verified to make sure that a given safety property holds.

All of the above mentioned uses of safety properties can be accomplished by specifying the properties as finite automata. While automata are useful in many cases, a more declarative approach, such as using a temporal logic, is usually preferred. Many model checking tools, such as Spin [2], support linear temporal logic (LTL).

In the automata theoretic approach to verification [3, 4, 5], LTL formulas are verified by translating their negation to Büchi automata, which are then synchronised with the system. If the synchronised system has an accepting execution, the property does not hold. One could benefit from using finite automata instead of Büchi automata if the given LTL property is a safety property. Reasoning about finite automata is simpler than reasoning about Büchi automata. For

---

explicit state model checkers, reasoning about Büchi automata requires slightly more complicated algorithms. In the symbolic context, emptiness checking with BDDs is in practice significantly slower than simple reachability [6]. For model checkers based on net unfoldings, such as [7], handling safety is much easier than full LTL [8].

Unfortunately, there are some complexity related challenges in translating LTL formulas to finite automata. A finite automaton specifying every finite violation of a LTL safety property can be doubly exponential in the size of the formula [1]. Formulas, for which every failing computation has an *informative* bad prefix, or alternatively called the non-pathological formulas, have singly exponential finite automata recognising their finite violations [1]. Deciding if an LTL formula is pathologic is a PSPACE-complete problem [1]. Pathological LTL formulas are not needed for expressiveness, as a pathological formula always can be expressed with an equivalent non-pathological one [1].

We present an efficient translation algorithm from LTL safety properties to finite automata based on [1]. The resulting finite automata can be used by explicit state model checkers and they can be fairly easily adapted to partial order semantics methods too [8]. The translation has been implemented in a tool and is experimentally evaluated. Experiments show that our approach is competitive and can result in significant gains when used. We have also implemented a decision procedure to decide if an LTL formula is pathologic. To our knowledge, this is the first time an implementation for checking if a formula is pathologic has been evaluated.

Other authors have also considered the problem of model checking of LTL safety properties. Kupferman and Vardi [1] present many complexity theoretical results and an algorithm on which the algorithm presented in this paper is based on. Some of the results are generalisations of results by Sistla [9]. Geilen [10] presents a tableau algorithm which essentially is a forward direction version of the algorithm of Kupferman and Vardi, which is described in the backward direction. Model checking safety properties expressed using past temporal operators has been considered at least by [11, 12].

The rest of this paper is structured as follows. Section 2 introduces the necessary theory. In Sect. 3 we present our translation from LTL to finite automata. We also discuss the relevant complexity issues. Section 4 covers issues related to implementation and the experiments are presented in Sect. 5. Finally, in Sect. 6 we discuss some implications of the results and consider avenues for further research.

2   Preliminaries

This section introduces the necessary theory and some notations presented in [1].

Let $w = \sigma_0\sigma_1\sigma_2\ldots \in \Sigma^\omega$ be an infinite word over the finite alphabet Σ. We denote the i:th position of the word by $w(i) = \sigma_i$ and the suffix $\sigma_i\sigma_{i+1}\ldots$ is denoted by w^i. A *Büchi* automaton is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, F \rangle$, where Σ is the *alphabet*, Q is a finite set of *states*, $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*,

$Q_0 \subseteq Q$ the set of *initial states*, and $F \subseteq Q$ is a set of *final states*. A *run* of the automaton $\mathcal{A}$ on a word $w \in \Sigma^\omega$ is a mapping $\rho : \mathbb{N} \to Q$ such that $\rho(0) \in Q_0$ and $(\rho(i), \sigma_i, \rho(i+1)) \in \delta$ for all $i \geq 0$. We use $inf(\rho)$ to denote the set of states occurring infinitely often in the run. A word w is accepted if it has a run ρ such that $inf(\rho) \cap F \neq \emptyset$. Finite automata differ from Büchi automata only in the acceptance condition. An automaton on finite words accepts a finite word $w = \sigma_0 \sigma_1 \ldots \sigma_n \in \Sigma^*$ if it has a run ρ on w such that $\rho(n+1) \in F$. The set of words accepted by an automaton $\mathcal{A}$, its *language*, is denoted $\mathcal{L}(\mathcal{A})$. Deciding if $\mathcal{L}(\mathcal{A}) = \emptyset$ is referred to as doing an emptiness check.

Let $L \subseteq \Sigma^\omega$ be a language on infinite words over an alphabet Σ. We say that a finite word $x \in \Sigma^*$ is a *bad prefix* for language L, if for every $y \in \Sigma^\omega$ we have that $x \cdot y \notin L$. Given a language L, if all $w \in \Sigma^\omega \setminus L$ have a bad prefix we call L a *safety language*. We denote the set of bad prefixes of a language L by $pref(L)$.

2.1   LTL

The syntax of LTL consists of atomic propositions, the normal boolean connectives, and *temporal operators*. Let AP be a set of atomic propositions. Well-formed formulae of LTL are constructed in the following way:

- **true**, **false** and every $p \in AP$ are well-formed formulae
- If ψ and φ are well-formed formulae, then so are $\psi \wedge \varphi$, $\psi \vee \varphi$, $\psi \, U \, \varphi$, $\psi \, V \, \varphi$, $\neg \varphi$ and $X \varphi$.

LTL is interpreted over infinite sequences of atomic propositions, i.e. infinite words in $(2^{AP})^\omega$. A model (or word) $\pi = \sigma_0 \sigma_1 \sigma_2 \ldots$, where $\sigma_i \subseteq AP$, is a mapping $\pi : \mathbb{N} \to 2^{AP}$. By π^i we denote the suffix $\pi^i = \sigma_i \sigma_{i+1} \sigma_{i+2} \ldots$ and π_i denotes the prefix $\pi_i = \sigma_0 \sigma_1 \ldots \sigma_i$. For an LTL formula ψ and a model π, we write $\pi^i \models \psi$, "the suffix π^i is a model of ψ". The semantics of the models relation $\models$ is defined inductively in the following way.

- For all π^i we have that $\pi^i \models$ **true** and $\pi^i \not\models$ **false**.
- For atomic propositions $p \in AP$, $\pi^i \models p$ iff $p \in \sigma_i$
- $\pi^i \models \psi_1 \vee \psi_2$ iff $\pi^i \models \psi_1$ or $\pi^i \models \psi_2$.
- $\pi^i \models \psi_1 \wedge \psi_2$ iff $\pi^i \models \psi_1$ and $\pi^i \models \psi_2$.
- $\pi^i \models X \psi$ iff $\pi^{i+1} \models \psi$.
- $\pi^i \models \neg \psi$ iff $\pi^i \not\models \psi$.
- $\pi^i \models \psi_1 \, U \, \psi_2$ iff there exists $k \geq i$ such that $\pi^k \models \psi_2$ and for all $i \leq j < k$ $\pi^j \models \psi_1$.
- $\pi^i \models \psi_1 \, V \, \psi_2$ iff for all $k \geq i$, if $\pi^k \not\models \psi_2$, then there is $i \leq j < k$ such that $\pi^j \models \psi_1$.

Usually we do not write $\pi^0 \models \psi$ but simply $\pi \models \psi$. LTL formulas which are in positive normal form (PNF) only have negations in front of atomic propositions. Any LTL formula can be rewritten into PNF using the duality between U and V. We use $cl(\psi)$ to denote the set of subformulas of ψ. The size of a formula, denoted $|\psi|$, is defined as the cardinality of $cl(\psi)$.

An LTL formula ψ specifies a language $\mathcal{L}(\psi) = \{\pi \in (2^{AP})^\omega \mid \pi \models \psi\}$. We say that an LTL formula is a safety formula if its language is a safety language.

3   Model Checking LTL Safety Properties

In the automata theoretic approach to LTL model checking the negation of the LTL specification is translated into a Büchi automaton $\mathcal{A}_{\neg\psi}$. The system is then viewed as an automaton and synchronised with the property automaton. If the property holds, the language of the synchronised automaton is empty.

Our goal is to construct a finite automaton which detects the bad prefixes for $\mathcal{L}(\psi)$. The resulting finite automata can be used e.g. for model checking, real-time monitoring [12] or as a specification for testing [13]. In the context of model checking treating safety as a special case has some benefits. One benefit is that fairness need not be taken into account, if we know we are dealing with a safety specification. Another benefit is that reasoning about finite automata is simpler. For instance, dealing with finite automata is much simpler than dealing with Büchi automata for model checkers based on net unfoldings [7]. For explicit state model checkers the algorithm for checking emptiness of Büchi is slightly more complicated than checking the emptiness for a finite automaton. When model checking with finite automata, we do not need to proceed in a depth-first order. Instead, we can e.g. apply a heuristic and do a best-first search to possibly obtain shorter counterexamples.

There are two major obstacles to using finite automata for LTL safety formulas. First of all, we must be able recognise safety formulas. This problem is unfortunately PSPACE-complete in the size of the formula [9]. A partial solution to this is that Sistla [9] has introduced a syntactic fragment of LTL which can only express safety properties. The fragment includes all LTL formulas which in PNF only contain the temporal operators V and X. The second problem is that translating an LTL safety formula to a finite automaton is hard in the general case. The worst case complexity of an automaton for $pref(\mathcal{L}(\psi))$ is doubly exponential in the size of ψ [1]. It turns out, however, that for well behaved formulas a singly exponential automaton is possible [1]. The notion of a well behaved formula is formalised through *informativeness*.

3.1   Informativeness

We consider LTL formulae in positive normal form. The notion of informativeness tries to formalise when a bad prefix for the formula can demonstrate completely why the formula failed. Let ψ be an LTL formula and π a finite computation $\pi = \sigma_0 \sigma_1 \ldots \sigma_n$. The computation π is *informative* for ψ iff there exists a mapping $L : \{0, \ldots, n+1\} \to 2^{cl(\neg\psi)}$ such that the following conditions hold:

- $\neg\psi \in L(0)$,
- $L(n+1)$ is empty, and
- for all $0 \leq i \leq n$ and $\varphi \in L(i)$, the following hold.
    - If φ is a propositional assertion, it is satisfied by σ_i.
    - If $\varphi = \varphi_1 \vee \varphi_2$ then $\varphi_1 \in L(i)$ or $\varphi_2 \in L(i)$.
    - If $\varphi = \varphi_1 \wedge \varphi_2$ then $\varphi_1 \in L(i)$ and $\varphi_2 \in L(i)$.
    - If $\varphi = X\varphi_1$, then $\varphi_1 \in L(i+1)$.

- If $\varphi = \varphi_1 \ U \ \varphi_2$ then $\varphi_2 \in L(i)$ or $[\varphi_1 \in L(i)$ and $\varphi_1 \ U \ \varphi_2 \in L(i+1)]$.
- If $\varphi = \varphi_1 \ V \ \varphi_2$ then $\varphi_2 \in L(i)$ and $[\varphi_1 \in L(i)$ or $\varphi_1 \ V \ \varphi_2 \in L(i+1)]$.

If π is informative for ψ, the mapping L is called the *witness* for $\neg\psi$ in π. Using the notion of informativeness, safety formulae can be classified into three different categories [1]. A safety formula ψ is *intentionally* safe iff all the bad prefixes for ψ are informative. The formula ψ is *accidentally* safe iff every computation that violates ψ has an informative prefix. In other words, ψ can have bad prefixes which are not informative. Every computation is, however, guaranteed to have at least one informative prefix. A safety formula ψ is *pathologically* safe if there is a computation that violates ψ and has no informative bad prefix.

Accidentally safe and pathologically safe formulas always contain redundancy. It is, however, an open problem if there are feasible ways to remove these redundancies. As previously mentioned, it is possible to construct a singly exponential finite automaton which detects all informative prefixes of a formula. This means that as long as the given formula is not pathologic, using this construct will return a correct result if used in model checking. For pathologic formulas we must either remove the redundancy, do model checking with a Büchi automaton, or use a doubly exponential construct. However, any counterexample returned by the singly exponential construction is a valid counterexample, even for pathological formulas.

Deciding if an LTL formula ψ is pathologic is a PSPACE-complete problem in the size of the formula [1]. The problem can be decided in the following way. It is possible to construct an alternating Büchi automaton $\mathcal{A}_\psi^{true}$ with a linear number of states in $|\psi|$ which accepts exactly all computations which have informative prefixes [1]. We can also construct an alternating Büchi automaton $\mathcal{A}_{\neg\psi}$ where $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\psi)$ [5]. Pathologic formulas have violating computations which are not informative. Thus, a formula is not pathologic if every computation that satisfies $\neg\psi$ is accepted by $\mathcal{A}_{\neg\psi}^{true}$ This can be verified by checking the containment of $\mathcal{L}(\mathcal{A}_{\neg\psi})$ in $\mathcal{L}(\mathcal{A}_{\neg\psi}^{true})$. The above check has the nice property that if ψ is not a safety formula it will automatically rejected as pathological. Thus, in our intended application there is no need for a separate check if ψ is a safety formula.

3.2   Translation Algorithm

The finite automaton for informative prefixes of Kupferman and Vardi [1] is suboptimal for explicit model checkers. It will almost always have a state for every subset of $cl(\psi)$. While Geilen's procedure [10] is not as inefficient, it will still produce big automata. A more efficient construction will only consider some subsets. We define the restricted closure $rcl(\psi)$ of a formula ψ as the smallest set with the following properties.

- All temporal subformulas $\varphi \in cl(\psi)$, i.e. formulas with a temporal operator at the root of their parse tree, belong to $rcl(\psi)$.
- If a formula $X\varphi$ belongs to $rcl(\psi)$ then $\varphi \in rcl(\psi)$.
- If no other rule applies, then the top-level formula ψ belongs to $rcl(\psi)$.

The restricted closure defines which sets of subformulas must be considered when constructing a finite automaton for an LTL formula. Temporal subformulas must belong to the restricted closure because they refer to other than the current state. There are two special cases when other formulas are also included. The first case is the immediate subformula of a next-operator. In this case the subformula must be kept to ensure that it will be true in the next state. The second case is when ψ is a propositional expression when the reason is that $rcl(\psi)$ cannot be empty, because this will result in an automaton with no states.

Let S be a subset of $cl(\psi)$. We define $sat(\psi, S)$ in the following way:

- $sat(\textbf{true}, S) = \textbf{true}$
- $sat(\textbf{false}, S) = \textbf{false}$
- $sat(\psi, S) = \textbf{true}$ if $\psi \in S$.
- $\psi = \psi_1 \vee \psi_2$: $sat(\psi, S) = \textbf{true}$ if $sat(\psi_1, S)$ or $sat(\psi_2, S)$.
- $\psi = \psi_1 \wedge \psi_2$: $sat(\psi, S) = \textbf{true}$ if $sat(\psi_1, S)$ and $sat(\psi_2, S)$.
- Otherwise $sat(\psi, S) = \textbf{false}$

We can now present our algorithm, which is an optimisation of the construction presented in [1]. The general idea of both algorithms is to start from an empty set of requirements and by going backwards compute all possible informative prefixes. The algorithm as it is presented here will produce an automaton where there are many transitions from one state to another state. In an implementation these arcs would of course be joined to conserve memory.

Input: A formula ψ in positive normal form.
Output: A finite automaton $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, F \rangle$.
proc translate(ψ)
$F := \{\emptyset\}; \Sigma := 2^{AP}$;
$Q := X := F$;
while($X \neq \emptyset$) **do**
    $S :=$"some set in X"; $X := X \setminus \{S\}$
      for each $\sigma \in 2^{AP}$ **do**
        $S' := \sigma$;
          for each $\varphi \in rcl(\psi)$ **do** //in increasing subformula order
            switch(φ) **begin**
                case $\varphi = \psi_1 \vee \psi_2$:
                if $(sat(\psi_1, S')$ or $sat(\psi_2, S'))$ **then** $S' := S' \cup \{\varphi\}$;
                case $\varphi = \psi_1 \wedge \psi_2$:
                if $(sat(\psi_1, S')$ and $sat(\psi_2, S'))$ **then** $S' := S' \cup \{\varphi\}$;
                case $\varphi = X\psi_1$:
                if $(\psi_1 \in S)$ **then** $S' := S' \cup \{\varphi\}$;
                case $\varphi = \psi_1 \; U \; \psi_2$:
                if $(sat(\psi_2, S')$ or $(sat(\psi_1, S')$ and $\varphi \in S))$ **then** $S' := S' \cup \{\varphi\}$;
                case $\varphi = \psi_1 \; V \; \psi_2$:
                if $(sat(\psi_2, S')$ and $(sat(\psi_1, S')$ or $\varphi \in S))$ **then** $S' := S' \cup \{\varphi\}$;
            end
            if $\sigma \notin rcl(\psi)$ **then** $S' := S' \setminus \{\sigma\}$;

```
        od
        if(sat(ψ, S')) then Q₀ := Q₀ ∪ {S'};
        δ = δ ∪ {(S', σ, S)};
        X := X ∪ {S'}; Q := Q ∪ {S'}
    od
od
```

The resulting non-deterministic automaton can easily be determinised. Although this can theoretically result in an exponential blow up, according to our experiments this does not usually occur. The deterministic automaton is, in fact, in most cases smaller than the original.

The correctness of the algorithm is quite easy to justify using the same arguments as Kupferman and Vardi [1]. There is technical report which considers these questions in more depth [14].

Theorem 1. *[14] Given an LTL formula ψ, the algorithm translate(ψ) constructs a finite automaton which accepts exactly the informative prefixes of $\neg\psi$.*

The theoretical bound achieved by our algorithm is in most cases better than the ones presented in [1, 10].

Theorem 2. *[14] The number states of the automaton is bounded by $2^{rcl(\psi)}$.*

For LTL formulas without the next-operator the bound is equal to $max(2^{tf(\psi)}, 2)$, where $tf(\psi)$ denotes the temporal subformulas of ψ.

4   Implementation

We have implemented the optimised translation algorithm for safety LTL formulae and also the check for determining if a formula is pathologic.

The implementation is BDD-based. BDDs are used to represent sets of formulas efficiently. Especially the translation algorithm heavily employs manipulation of sets, which can easily be implemented with BDDs. However, BDDs can also incur a certain overhead making the algorithm slower in some cases compared to algorithms using simpler set representations.

The tool, *scheck*, has been implemented using ANSI C++ and it should compile on most platforms where a C++-compiler supporting templates is available. It is available online from http://www.tcs.hut.fi/~timo/scheck under the terms of the GNU general public license (GPL). It easy to use *scheck* with Spin. *scheck* uses the neverclaim facility together with "-DSAFETY" and a failing assertion to stop Spin when a counterexample has been found.

The implementation of the translation algorithm is split into four separate stages. The first stage simply parses the input formula and transforms it into positive normal form. Optionally it can also perform some simple checks such as check for syntactic safety of the formula. The next stage builds a symbolic transition relation characterising the given formula. The third stage optionally performs some automata theoretic transformations, such as determinisation of

the automaton. The fourth and the last stage outputs the automaton to the desired file or stream.

The basic idea of the second stage is to construct a symbolic transition relation which adheres to the translation rules given in the previous section. Symbolic reachability analysis is used to construct the automaton. To represent the states, $2 * N$ BDD variables are reserved, where N is the number of subformulas, i.e. $|cl(\psi)| = N$. One variable describes if the subformula belongs to the current state and one variable is for the next state. By using quantification it is easy remove state bits not in $rcl(\psi)$.

The third stage of the translation is an optional determinisation of the automaton. Early experiments showed that in almost all cases determinisation makes the automaton smaller. A deterministic automaton also has shorter model checking times, because it causes less branching in the product automaton. See the section on experiments for more details. If the automata are to be used for monitoring executions of software, determinisation is mandatory, although it can be performed on-the-fly while monitoring. Before the third stage, the automaton is converted to an explicit representation. Determinisation is easier when the automaton is in an explicit form. The arcs are still represented as BDDs since this allows easy manipulation of the arcs.

The last stage of the translation outputs the automaton to a file or a stream. Here, the only challenge is to output the arc labelling, represented as BDDs, succinctly using $\wedge$, $\vee$ and negation in front of the propositions.

4.1   Checking Pathologic Safety

Implementing, a check for if a formula is pathologic involves implementing an emptiness check for the intersection of two automata. Recall that an LTL formula ψ is pathologic iff $\mathcal{L}(\mathcal{A}_{\neg\psi}) \not\subseteq \mathcal{L}(\mathcal{A}_{\neg\psi}^{true})$. This is equivalent to that $\mathcal{L}(\mathcal{A}_{\neg\psi} \times \bar{\mathcal{A}}_{\neg\psi}^{true}) \neq \emptyset$.

Our implementation does not directly follow the procedure described in [1]. We perform the following steps when we are given an LTL formula ψ.

1. Construct a Büchi $\mathcal{A}_{\neg\psi}$ automaton corresponding to the negation of ψ.
2. Construct a *deterministic* finite automaton $\mathcal{B}_{\neg\psi}$, which accepts all informative bad prefixes of ψ.
3. Interpret $\mathcal{B}_{\neg\psi}$ as a deterministic Büchi automaton and construct the complement $\bar{\mathcal{B}}_{\neg\psi}$.
4. Construct the product automaton $\mathcal{C} = \mathcal{A}_{\neg\psi} \times \bar{\mathcal{B}}_{\neg\psi}$.
5. Check if $\mathcal{L}(\mathcal{C}) = \emptyset$.

The reason we require that $\mathcal{B}_{\neg\psi}$ is deterministic is that complementing a nondeterministic Büchi automaton is complicated and has an exponential time lower bound [15], while complementing a deterministic Büchi automaton can be done in linear time. The procedure outlined above is not optimal in the complexity theoretical sense but it works quite well when the size of $\mathcal{B}_{\neg\psi}$ does not explode. An optimal approach could use alternating automata, as outlined in [1].

We have presented how all steps can be performed except the complementation of the deterministic Büchi automaton. We follow the presentation Vardi given in his lecture notes [16]. Let $\mathcal{A} = \langle \Sigma, Q, \delta, s_0, F \rangle$ be a deterministic Büchi automaton. The complement $\bar{A} = \langle \Sigma, \bar{Q}, \bar{\delta}, \bar{s_0}, \bar{F} \rangle$ can be computed with the following operations.

- $\bar{Q} = (Q \times \{0\}) \cup ((Q - F) \times \{1\})$,
- $\bar{s_0} = s_0 \times \{0\}$,
- $\bar{F} = (S - F) \times \{1\}$, and
- for all states $q \in Q$ and symbols $a \in \Sigma$:

$$\bar{\delta}((q,0),a) = \begin{cases} \{(\delta(q,a),0)\}, & \text{if } \delta(q,a) \in F \\ \{(\delta(q,a),0),(\delta(q,a),1)\}, & \text{if } \delta(q,a) \notin F \end{cases}$$

$$\bar{\delta}((q,1),a) = \{(\delta(q,a),1)\}, \delta(q,a) \notin F$$

The size of the complement is at most twice the size of the original automaton.

In the implementation we first compute explicit state representations of $\mathcal{A}_{\neg\psi}$ and $\mathcal{B}_{\neg\psi}$. Next, the deterministic automaton $\mathcal{B}_{\neg\psi}$ is complemented using the procedure above. Finally the product is computed and an emptiness check is performed using Tarjan's algorithm for finding strongly connected components. The tool has an interface for using an external translator to construct the Büchi automaton $\mathcal{A}_{\neg\psi}$, to benefit from more optimised Büchi translators than the simple one of the tool.

5   Experiments

In order to evaluate the implementation, *scheck*, we conducted some experiments. Four experiments were performed. The two first experiments measured the performance of the tool for random formulae, while for the third experiment we used the formulas in the specification pattern system [17]. In the fourth experiment *scheck* was interfaced with the model checker Spin to measure performance on practical models. We have collected the models used and other relevant files to a webpage: `http://www.tcs.hut.fi/~timo/spin2003`.

We used three LTL to Büchi translators as reference: a state of the art tool by Paul Gastin and Dennis Oddoux [18], the translator packaged with the Spin tool [2], and an efficient implementation of the algorithm described in [19] by Mäkelä, Tauriainen and Rönkkö [20]. In the following we refer to the tool of Gastin and Oddoux as *ltl2ba*, to the tool of Mäkelä et al. as *lbt* and to the translator of Spin simply as *spin*. The two first tests were conducted on a machine with a 266 MHz Pentium II processor with 128 MB of memory. The third and fourth tests were conducted on a machine with a 1 GHz AMD Athlon processor with 1 GB of memory.

For the two first tests which involve random formulae and random statespaces we have used the LTL to Büchi translator test bench by Tauriainen and Heljanko [21]. The tool includes facilities for randomly generating LTL formulae

and measuring different statistics such as the size of the generated automaton and generation time.

The first test generates random syntactically safe formulae. Most safety formula encountered in practice will probably be of this form. The idea is to measure how well the tools can cope with typical safety formulae. Statistics measured are the number of states and transitions in the automata produced, the time to generate the automata and the size of the product of a random state space of twenty states and the automaton. The number states and transitions in the generated automaton and generation give an indication of the general performance of the translator while the size of the product statespace depends on both the size of the generated automaton and the structure of the automaton. Automata which have small product statespaces can potentially at an early stage 'decide' if the current sequence under inspection cannot satisfy the given formula.

We generated one thousand formulas of fixed length between five and 22, three times. For *lbt* and *spin* we stopped the generation at 15 because we started to run out of memory. We set *scheck* to generate deterministic automata, as preliminary experiments indicated that this improved performance. To compare to the other tools we computed the mean $E(M(proc, L))$ over the three times for each procedure *proc*, formula length L and measure M. The ratio of the means $\frac{E(M(scheck,L))}{E(M(proc,L))}$ have been computed in Figure 1.

When we compare the size of the automatons generated, i.e. number of states and transitions, *scheck* seems to be very competitive when formula sizes grow. Especially the procedures based on [19] cannot compete well. When the formulas are short *spin* and *ltl2ba* are able to compete, but when the length of the formulas grows, *scheck* clearly scales better than the other tools. At the time when the measurements were made *scheck* did not check for a "sink state" in its deterministic automata. If the tests were rerun, *scheck* would probably narrow down the small lead *spin* and *ltl2ba* have in short formulae. Long formulae are not affected as much by the removal of one sink state. Note that in the number of transitions *scheck* scales even better compared to the other tools. One reason is probably that *scheck* generates deterministic automata.

Generation time gives a different picture of how well the tools perform. The tools based on [19] have an advantage with short formulae but do not scale as well. *ltl2ba* is however much faster than *scheck* in all cases. It scales better and it is faster for short formulae. It is possible that the implementation of *scheck* using BDDs gives the other tools an competitive advantage. This is supported by the fact that *scheck* produces smaller or as small automata as the other tools but is still in some cases significantly slower.

We expected *scheck* to have smaller product statespaces, because the automata it generates are deterministic. When the automaton is deterministic, the branching factor should be smaller. The results also confirm this. *scheck* generates smaller product statespaces than all three other tools.

The second test is in a sense a generalisation of the first. Now we randomly generate any type of LTL formula and use the implemented check for pathologic formulae to see if its a safety formula which can be used in the tests. This test
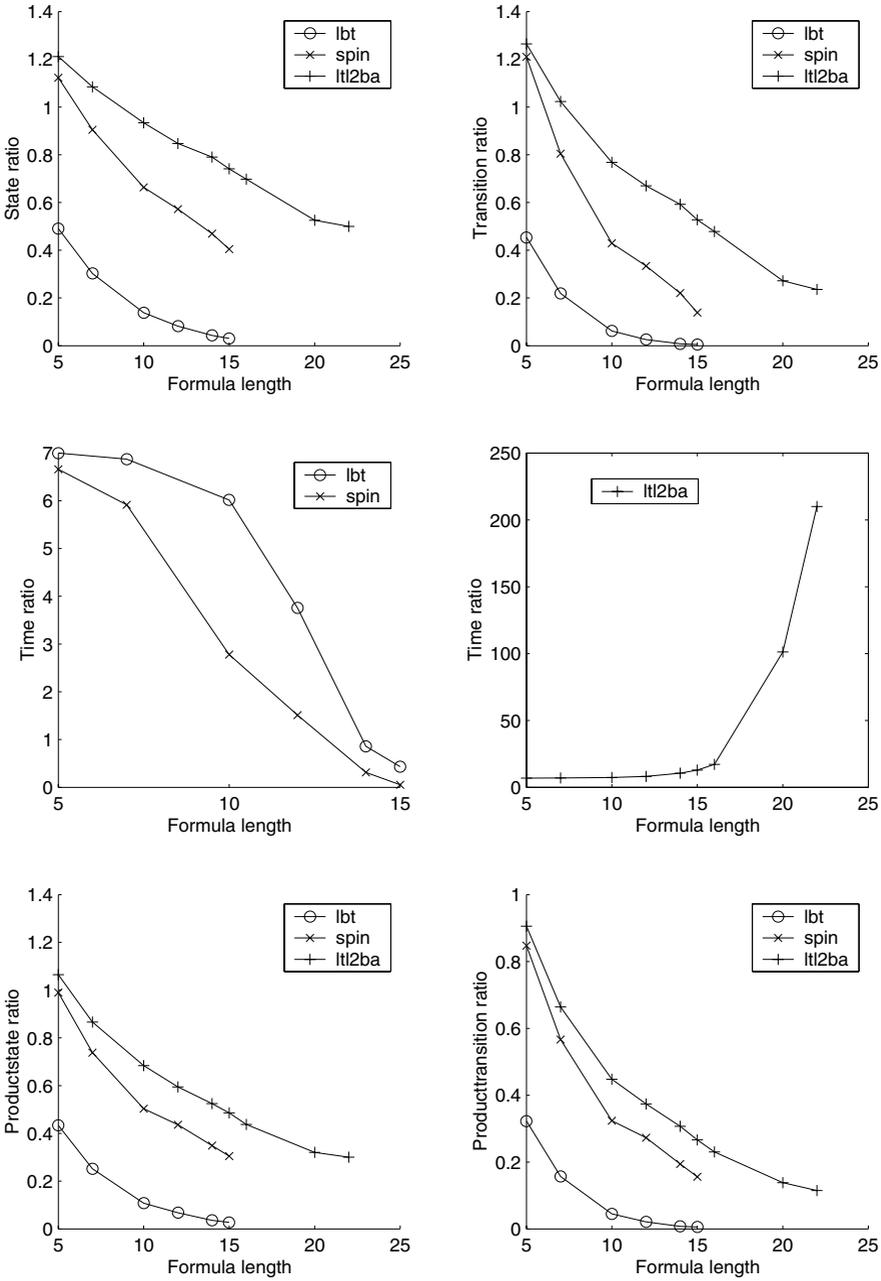
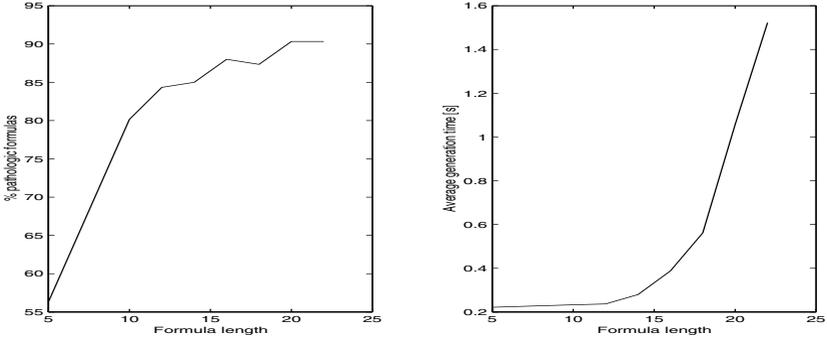**Fig. 1.** Comparison of the tools with syntactically safe formulae

Fig. 2. *scheck* performance for general formulae

is only performed for *scheck*, as none of the other tools can check if a formula is pathologic.

One hundred formulas and their negation were generated for each length ranging from five to 22. As can been seen from the first plot, most of the generated pathologic or liveness formulas and the percentage grows when formula length increases. This is of course not surprising as the temporal operators often describing liveness properties are more likely to occur. The generation time, which includes the time to check if the formula is pathologic, for formulas which are not pathologic shows the familiar exponential increase which usually manifests itself sooner or later when solving PSPACE-complete problems. Our suspicion is that why *scheck* can require exponential time but not generate exponential automata is due to inefficiencies in implementation when using BDDs in *scheck*. One conclusion which is not affected by high rejection ratio of the formulas is that *scheck* can clearly scale well when identifying pathologic formulas. We refer the interested reader to [14] for more statistics.

The third test measures the same statistics as the two first but here we use the formulas from the specification pattern system [17]. Of the 60 patterns for LTL, 36 patterns are safety formulas to which we applied the translators. As *spin* ran out of memory for some of the formulas it was excluded from the test. The results can be found in Table 1.

For the fourth test we used five models of distributed algorithms. We used the model checker Spin, using "-DSAFETY" with *scheck* and normal LTL model checking with the others, to check a safety property on each model. Because the

Table 1. Statistics for the specification pattern formulas (36 formulas)

	states	arcs	time [s]	product states	product arcs
ltl2ba	160	348	0.5	3037	15406
lbt	1915	31821	1.2	25134	763203
scheck	144	316	2.1	2481	9806

Table 2. Experiments on practical models. A '-' means out of memory

model	scheck			spin			ltl2ba			
	states	arcs	t [s]	states	arcs	t [s]	states	arcs	t [s]	$\|\psi\|$
peterson(3)	17476	32343	0.06	21792	45870	0.09	21792	45870	0.09	10
peterson(4)	3254110	709846	20.8	4216030	10315000	37.3	4216030	10315000	37.5	10
sliding(1,1)	130799	407238	0.9	258456	890026	2.2	258432	890386	2.2	21
sliding(1,2)	518050	1670120	3.9	1027130	3604660	9.8	1027120	3604410	9.8	21
sliding(2,1)	5447700	18271400	534.7	10794100	39649800	1097.4	10794000	39645700	1097.6	21
erathostenes(50,1)	522	522	0.03	522	522	0.03	678	678	0.03	7
erathostenes(60,2)	324	324	0.02	357958	647081	4.0	794322	1319710	8.4	10
erathostenes(70,3)	522	522	0.04	2047030	4407400	48.5	3110700	6474410	76.6	13
erathostenes(80,4)	789	789	0.04	-	-	-	-	-	-	16
erathostenes(80,5)	847	847	0.04	-	-	-	-	-	-	19
iprot	7095180	20595400	377.0	16011900	46288600	1006.2	16011900	46288600	1003.7	21
giop	146646	215640	1.8	255105	524493	4.8	255105	524493	4.6	9

tool *lbt* did not have a Spin interface it did not participate in the tests. The property for *peterson*, *sliding*, *iprot*, and *giop* is a property which holds while the property for *erathotenes* does not hold. Additionally, the specification for *erathotenes* was parametric (the second parameter of the model). The statistics compared are size of the product statespace, which gives an indication of the memory use of the model checker, and the time used for model checking. Table 2 contains the results.

It would seem that the advantage of using finite automata for safety properties is very significant when debugging models. The fact that the algorithm can stop immediately when the error has been discovered without completing an infinite loop can result in significant gains. If the property holds the gain is smaller, but still there. It would appear that *scheck* causes less branching in product statespace. These results are of course preliminary and to get a clearer picture of the situation more tests are required.

6   Discussion

The implementation of the translation procedure presented in this work, *scheck*, produces smaller automata than the state of the art of the LTL to Büchi automata translators. In some cases the difference is exponential, while in other times it is negligible. The resulting product statespaces are also smaller for *scheck*. This is probably because *scheck* produces deterministic automata. The fact that determinising would result in much smaller automata came as a pleasant surprise. It is a well-known that determinising a non-deterministic automaton can result in an exponentially larger automaton. Safety properties can be expressed using reverse deterministic automata [1]. Apparently, the succinctness of non-determinism is not needed in the forward direction, when using random formulae. Currently *scheck* (version 1.0) does not apply a minimisation algorithm to the deterministic automata produced. This would be simple to add but we leave this to further work. Automata generation time was the one area where the results for *scheck* were disappointing. Although *scheck* generates the automaton for almost any formula used the tests in a few seconds, this is quite slow compared ltl2ba which in most cases would only use a few hundreths of a second. One of the reasons could be that *scheck* uses BDDs to manage sets,

which sometimes can cause overhead even when the sets are small. A non-BDD implementation would probably perform better. To produce even smaller automata faster than *scheck*, another approach is probably required. It would be interesting to see if starting from alternating automata as in [18] could facilitate an efficient translation.

For practical models it would seem that using *scheck* gives the most gain when debugging models, especially if the properties are complex. The gain will not manifest itself if we are model checking a formula of the form $\mathbf{G}p$ as most LTL to Büchi automata translators handle this case optimally. More experiments are needed to get a clearer picture of the situation.

In order to be able to benefit from treating safety properties as a special case we must be able to recognise safety formulae. There are two ways in which this can be done. Either we only use the syntactically safe subset of LTL, which is easy to recognise or we implement pathologic checking in the tool. Experiments seem to confirm that pathologic checking is feasible. This means that both options for recognising safety properties are available and can be used. In some cases, as many probably have noticed, it is possible to interpret the Büchi automaton produced by a normal translator as a finite automaton, and use it directly for model checking. This could be confirmed by interpreting the produced finite automaton as the output of *scheck* and check if it is pathologic. In this way, a normal LTL to Büchi automata translator could in some cases be used to output finite automata. The construction of Geilen [10] has the advantage that it can provably produce correct Büchi and finite automata for informative prefixes with very small changes.

The results of Geilen [10] indicate that there is tight relationship between the Büchi automata and the finite automata for LTL formulas. In the future it would seem reasonable that a translator tool could produce both small finite automata and small Büchi automata.

scheck is available online from `http://www.tcs.hut.fi/~timo/scheck` and ready to be used with Spin.

Acknowledgements

References

[1] Kupferman, O., Vardi, M.: Model checking of safety properties. Formal Methods in System Design **19** (2001) 291–314   74, 75, 77, 78, 79, 80, 81, 86
[2] Holzmann, G.: The model checker Spin. IEEE Transactions on Software Engineering **23** (1997) 279–295   74, 82
[3] Kurshan, R.:   Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeon University Press (1994)   74
[4] Vardi, M. Y., Wolper, P.: Automata-theoretic techniques for modal logic of programs. Journal of Computer and System Sciences **32** (1986) 183–221   74

[5] Vardi, M.: An automata-theoretic approach to linear temporal logic. In: Logics for Concurrency: Structure versus Automata. Volume 1043 of LNCS. Springer (1996) 238–266   74, 78

[6] Hardin, R., Kurshan, R., Shukla, S., Vardi, M.: A new heuristic for bad cycle detection using BDDs. In: Computer Aided Verification (CAV'97). Volume 1254 of LNCS., Springer (1997) 268–278   75

[7] Esparza, J., Heljanko, K.: Implementing LTL model checking with net unfoldings. In: SPIN 2001. Volume 2057 of LNCS., Springer (2001) 37–56   75, 77

[8] Heljanko, K.: Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets. PhD thesis, Helsinki University of Technology, Department of Computer Science and Engineering (2002)   75

[9] Sistla, A.: Safety, liveness, and fairness in temporal logic. Formal Aspects in Computing **6** (1994) 495–511   75, 77

[10] Geilen, M.: On the construction of monitors for temporal logic properties. In: RV'01 - First Workshop on Runtime Verification. Volume 55 of Electronic Notes in Theoretical Computer Science., Elsevier Science Publishers (2001)   75, 78, 80, 87

[11] Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verication. Model-Checking Techniques and Tools. Springer (2001)   75

[12] Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 2280 of LNCS., Springer (2002) 342–356   75, 77

[13] Helovuo, J., Leppänen, S.: Exploration testing. In: Application of Concurrency in System Design (ACSD'2001), IEEE (2001) 201–210   77

[14] Latvala, T.: On model checking safety properties. Technical Report   80, 85 HUT-TCS-A76, Helsinki University of Technology (2002) Available from http://www.tcs.hut.fi/Publications

[15] Safra, S.: Complexity of Automata on Infinite Objects. PhD thesis, The Weizmann Institute of Science (1989)   81

[16] Vardi, M.: Automata-theoretic approach to design verification. Webpage (1999) http://www.wisdom.weizmann.ac.il/~vardi/av/notes/lec2.ps   82

[17] Dwyer, M., Avrunin, G., Corbett, J.: Property specification patterns for finite-state verification. In: Workshop on Formal Methods in Software Practice, ACM Press (1998) 7–15   82, 85

[18] Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Computer Aided Verification (CAV'2001). Volume 2102 of LNCS., Springer (2001) 53–65   82, 87

[19] Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Protocol Specification, Testing, and Verification, Warsaw, Chapman & Hall (1995) 3–18   82, 83

[20] Mäkelä, M., Tauriainen, H., Rönkkö, M.: lbt: LTL to Büchi conversion (2001) http://www.tcs.hut.fi/Software/maria/tools/lbt/   82

[21] Tauriainen, H., Heljanko, K.: Testing LTL formula translation into Büchi automata. STTT - International Journal on Software Tools for Technology Transfer **4** (2002) 57–70   82